

United States Air Force Program Office
Guide to Ada, Edition 3

CHRISTINE AUSNIT
ERNESTO GUERRIERI
NANCY INGWERSEN
SUZANNE RUEGSEGGER

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02254

31 December 1987

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

Prepared For

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
DEPUTY FOR ADVANCED DECISION SYSTEMS
HANSCom AIR FORCE BASE, MASSACHUSETTS 01731



ADA189651

LEGAL NOTICE

When U. S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

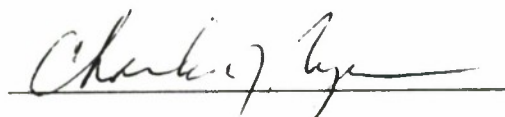
OTHER NOTICES

Do not return this copy. Retain or destroy.

This technical report has been reviewed and is approved for publication.

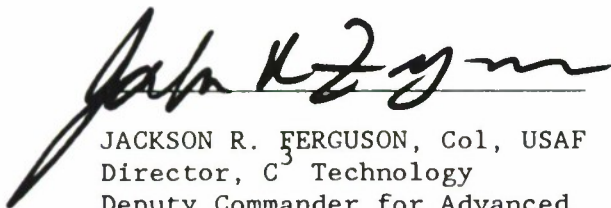


MARK V. ZIEMBA, 1Lt, USAF
Project Manager, Project 2526
Computer Resource Management
Technology Program (PE 64740F)



CHARLES J. RYAN, Maj, USAF
Program Manager, Computer Resource Management
Technology Program (PE 64740F)
Deputy Commander for Advanced Decision System

FOR THE COMMANDER



JACKSON R. FERGUSON, Col, USAF
Director, C³ Technology
Deputy Commander for Advanced
Decision Systems

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

REPORT SECURITY CLASSIFICATION Unclassified		1b. RESTRICTIVE MARKINGS	
SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for Public Release; Distribution Unlimited	
DECLASSIFICATION / DOWNGRADING SCHEDULE			
PERFORMING ORGANIZATION REPORT NUMBER(S) 3451-4-010/2		5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-88-102	
NAME OF PERFORMING ORGANIZATION SofTech, Inc.	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION HQ Electronic Systems Division (AVSE)	
ADDRESS (City, State, and ZIP Code) 460 Totten Pond Road Waltham MA 02254		7b. ADDRESS (City, State, and ZIP Code) Hanscom AFB Massachusetts, 01731-5000	
NAME OF FUNDING / SPONSORING ORGANIZATION Deputy for Advanced Decision Systems	8b. OFFICE SYMBOL (If applicable) ESD/AVSE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F33600-87-D-0337	
ADDRESS (City, State, and ZIP Code) Hanscom AFB Massachusetts, 01731-5000		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO.	PROJECT NO.
		TASK NO.	WORK UNIT ACCESSION NO.
TITLE (Include Security Classification) United States Air Force Program Office Guide to Ada, Edition 3			
PERSONAL AUTHOR(S) Christine Ausnit, Ernesto Guerrieri, Nancy Ingwersen, Suzanne Ruegsegger			
TYPE OF REPORT Technical	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1987 December 31	15. PAGE COUNT 82
SUPPLEMENTARY NOTATION			
COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	
		Ada, interfaces, tools, verification, Computer Aided Software Engineering (CASE), ADETS	
ABSTRACT (Continue on reverse if necessary and identify by block number)			
<p>The purpose of the Program Office Guide to Ada is to discuss issues affecting the selection development and maintenance of systems whose software is written in the Ada language. Each edition focuses on a different set of topics and their implications for managers.</p> <p>This edition concentrates on: Maintenance and revision of the Ada standard, Ada Education and Training Study (ADETS), program proving and verification, environments, tools, interfaces and Computer Aided Software Engineering.</p>			
DISTRIBUTION / AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
NAME OF RESPONSIBLE INDIVIDUAL M.V. Ziemba, 1Lt, USAF		22b. TELEPHONE (Include Area Code) (617) 377-2656	22c. OFFICE SYMBOL ESD/AVSE

EXECUTIVE SUMMARY

This report is the third of four editions for the Program Office Guide to Ada. It complements the Program Manager's Guide to Ada, ESD-TR-85-159, dated May 1985. This effort was sponsored by the Air Force Computer Resource Management Technology Program, Program Element 64740F, Project 2526, Software Engineering Tools and Methods.

The first edition addressed the following topics: policy, run-time efficiency, run-time support environment customization, training, Ada program design languages, and conversion of non-Ada code.

The second edition addressed the following topics: DoD Standards 2167 and 2168, guidelines for proposal evaluation, reusability and portability considerations, software costing models, benchmarking efforts, and Ada software libraries.

The third edition focuses on:

- Maintenance and revision of the Ada Standard,
- Ada Education and Training Study,
- Program Proving and Verification,
- Environments, Tools, Interfaces, and
- Computer-Aided Software Engineering.

The Ada Standard will soon have existed for five years. According to typical standardization procedures, it is expected that the Ada Standard will be reviewed in the near future. This review could recommend a revision to the language sometime in the 1990s. In the meantime, procedures are in place to maintain the Ada Standard.

It was noted in Edition 1 that the Armed Forces Communications and Electronics Association had undertaken a study of Ada education and training. After extensive data collection and analysis, this report has been published and its findings are discussed.

Formal methods, program verification in particular, are an expensive and intellectually demanding field. Much work has been done in terms of developing proof rules for Ada language constructs. Even when verification of a full program is impractical, it is feasible to verify a limited set of properties implemented in a "verifiable" subset of Ada constructs.

Numerous software development aids exist and are being developed for Ada. These include programming environments, tools (either as part of an environment or stand-alone), and interfaces to other programming devices, such as databases. Tool focus is

increasingly on software engineering and programming support tools, as opposed to the earlier emphasis on program development tools.

FOREWORD

This report is the third in a series of four editions that supplement the Program Manager's Guide to Ada, ESD-TR-85-159, published by The Computer Resource Management Technology Program in May, 1985. The introduction of Ada as the mandated high order language for Mission Critical Computer Programming in the Department of Defense has generated a need for clear, concise information for Program Managers and others concerned with cost, schedule, and performance in the application of this new language.

The intent of this series is to bring Program Office personnel up to date on facts presented in the original Program Manager's Guide, as well as to provide a more rounded discussion on certain subjects presented in the original guide. This series of four reports is designed for the Program Manager and his technical staff. It is recommended that this report be kept with the original Program Manager's Guide to Ada, forming a ready reference to Ada and Ada-related topics.

ACKNOWLEDGEMENTS

This report is sponsored by the Air Force Computer Resource Management Technology Program, PE 64740F, Project 2526 (Software Engineering Tools and Methods), ESD/XRSE, Hanscom Air Force Base, Massachusetts.

The Computer Resource Management Technology Program is the Air Force engineering development program to develop and transfer into active use the technology, tools, and techniques needed to cope with the explosive growth in Air Force systems that use computer resources. The goals of the program are to: (a) provide for the transition of computer system developments in laboratories, industry, and academia to Air Force systems; (b) develop and apply software acquisition management techniques to reduce life-cycle costs; (c) provide improved software design tools; (d) address the various problems associated with computer security; (e) develop advanced software engineering tools, techniques, and systems; (f) support the implementation of high order languages, e.g., Ada; (g) address human engineering for computer systems; and (h) develop and apply computer simulation techniques for the acquisition process.

Contents

Executive Summary	iii
Foreword	v
Acknowledgements	vi
15 Impact of Revisions to the Ada Standard	15-1
15.1 The Various Ada Standards	15-1
15.2 The Standardization Process for the Ada Language	15-1
15.3 The Maintenance of the Ada Standard	15-2
15.3.1 The Maintenance Process	15-2
15.3.2 Impact of Maintenance	15-3
15.4 The Revision of the Ada Standard	15-4
16 AFCEA Ada Education Training Study (ADETS)	16-1
16.1 Purpose of Study	16-1
16.1.1 Study Team Composition: Industry	16-1
16.1.2 Definition of Effort	16-1
16.2 Sources of Data	16-2
16.3 Principal Findings	16-2
16.3.1 Management Commitment	16-3
16.3.2 Training Shortfalls	16-3
16.3.3 Number of Trained Personnel	16-3
16.4 Study Recommendations	16-4
17 Program Proving and Verification	17-1
17.1 Overview	17-1
17.1.1 Program Language Semantics	17-2
17.1.2 Verification Process	17-3
17.2 Program Testing and Verification	17-4
17.3 Practicality of verification	17-5

17.3.1	Benefits	17-5
17.3.2	Difficulties and costs	17-6
17.4	Cost reduction approaches	17-8
17.4.1	IBM Cleanroom Technique	17-9
17.4.2	Selective Use of Verification	17-10
17.4.3	Verifiable Ada Subset	17-10
17.5	Summary of current research	17-11
17.5.1	Full Semantic Specification of Ada	17-11
17.5.2	PolyAnna	17-12
17.5.3	Verifiable Subset Definition	17-12
17.5.4	Security Analysis of Ada Programs	17-12
17.5.5	Organizations on Formal Methods	17-13
18	Environments	18-1
18.1	Stoneman	18-1
18.1.1	Definition and Purpose	18-1
18.1.2	Structure	18-2
18.1.3	Current Use of Stoneman	18-5
18.2	CAIS	18-5
18.2.1	Progress on CAIS	18-6
18.2.2	Comparison to Stoneman	18-7
18.2.3	CAIS Implementation Validation	18-7
18.3	SDME	18-7
18.3.1	Structure	18-8
18.4	STARS - Environment Products	18-9
18.5	Software Life Cycle Support Environment	18-10
18.5.1	SLCSE Tools	18-11
19	Tools	19-1
19.1	Interactive Ada Workstation	19-1
19.1.1	Description and Purpose	19-1

19.1.2	Structure	19-2
19.1.3	Current Status	19-3
19.1.4	Comparison to Stoneman	19-4
19.2	STARS - Technology Development	19-4
19.3	Requirements to PDL Tool	19-5
20	Interfaces	20-1
20.1	Ada Approach to Interfaces	20-1
20.1.1	History and Rationale	20-1
20.1.2	Proliferation vs. Standardization	20-2
20.2	DBMSs and Ada	20-3
20.2.1	Description and Purpose	20-3
20.2.2	Ada/SQL Proposed Binding	20-4
20.2.3	Current Ada/DBMS Efforts	20-4
20.3	4GLs and Ada	20-5
20.3.1	Program Generators	20-5
20.3.2	Artificial Intelligence and Ada	20-6
20.4	Graphics and Ada	20-8
20.5	Other efforts	20-8
21	CASE (Computer-Aided Software Engineering)	21-1
21.1	Description and Purpose	21-1
21.2	Impact on Ada	21-3
21.3	Environment Standards and CASE	21-5
21.4	CASE Activities	21-6
A	Appendix: References	A-1
B	Appendix: Bibliography	B-1
C	Appendix: Points of Contact for Ada Information	C-1

Section 15

Impact of Revisions to the Ada Standard

The importance of coordinating standards in programming languages arises from the need to transfer programs from one computer installation to another in a different domain. Currently there are three standards for the Ada Language. There is interaction between the standards organizations to aid in keeping the Ada Standard consistent and usable.

When a standard is created or revised, the transition from one version of the standard to another occurs instantaneously as far as the standardization body is concerned. However, manufacturers cannot be expected to provide a new compiler overnight or discard a compiler that is based on the superseded standard. Consequently, there is a protracted transition from one standard to the next. This section will cover the current standardization process for the Ada language and its impact on programs utilizing the Ada language.

15.1 The Various Ada Standards

In 1983, Ada was adopted as both an American National Standard and a Military Standard (ANSI/MIL-STD-1815A). The Ada Standard was also adopted as an International Standard by the International Standards Organization (ISO) in 1987¹. This reflects the adoption of the Ada programming language in different domains (i.e., National, Military, and International, respectively). For Department of Defense (DoD) programs, the relevant Ada Standard is MIL-STD-1815A.

15.2 The Standardization Process for the Ada Language

After the initial development of a standard for the Ada language, the standardization process consists of two principal activities:

- the maintenance of the Ada Standard, and
- the revision of the Ada Standard.

The maintenance of the Ada Standard consists of interpreting the standard when the standard is unclear or ambiguous. Maintenance is an ongoing process, whereas revision takes place at clearly defined time intervals and could involve more major changes to the language. Sections 15.3 and 15.4 discuss both processes in depth.

¹ISO standards invariably, and probably inevitably, take longer to be produced than national standards.

15.3 The Maintenance of the Ada Standard

The maintenance of the Ada Standard consists of responding to issues about the language. The issues consist of comments about unclear or ambiguous parts of the standard (as well as misunderstandings about the standard). These issues are known as commentaries.

15.3.1 The Maintenance Process

The Ada Rapporteur Group (ARG), a committee established by the ISO/TC 97/SC 22/WG 9, prepares the commentaries. The Working Group 9 (WG 9) reviews them and may approve them. The WG9 has been authorized to produce a technical report addressing all the issues and questions on the Ada language and the corresponding commentaries issued by the ARG. This report, however, will have no official standing as a maintenance document.

In discussing the maintenance process, it is important to distinguish the role of commentaries in the three bodies concerned with the Ada Standard, namely the ISO, the AJPO, and ANSI. Neither ISO nor ANSI recognize the commentaries as official Ada interpretations, even though they are prepared by WG 9. The commentaries may, however, have an effect on the MIL-STD in so far as they have an official standing with respect to the validation tests (Ada Compiler Validation Capability or ACVC). The organizations concerned with the standardization of the Ada language are currently discussing the maintenance process and should come to some agreement soon.

The AJPO requires that compilers pass the ACVC suite in order to conform to the MIL-STD. The effect of the commentaries is visible in the process discussed below.

The AJPO presents the WG 9 approved commentaries to the Ada Board for consideration. Should the Ada Board recommend their approval, then the Director of the AJPO may also approve them. It is this last approval which gives the commentaries their standing as official interpretations of the Ada Language with respect to the validation tests. The immediate impact of an "official" commentary is to resolve disputes about tests which address the same issue(s) as the commentary. The long term effect of official commentaries is discussed in Section 15.3.2.

To address a problem with the Ada Standard, a user can submit a comment on the Ada Standard. These comments will then be incorporated into the set of known issues about the Ada Standard, commentaries, and interpretations of these issues will be made by the appropriate standardization bodies. The comments can be submitted in writing to either the AJPO, the Ada Validation Office (AVO), or, via MILNET, to ada-comm@AJPO.SEI.CMU.EDU. The AVO may be contacted at:

Audrey Hook
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria, VA 22311
(703) 824-5501

There also exist forums in which users can raise and discuss issues concerning the Ada Standard. One of these is the Ada Language Issues Working Group (ALIWG), a working group under the Users Committee of ACM SIGAda. The ALIWG charter is to "provide a forum for discussion and review of the Ada language definition. Any recommendation supported by the group will be submitted to the AJPO."

15.3.2 Impact of Maintenance

Recent interpretations of the standard have included a wide variety of areas. These include:

- numeric issues,
- string and aggregate issues,
- parameter passing issues,
- I/O issues,
- compilation unit issues,
- issues on static expressions and subtypes,
- overloading resolution issues,
- visibility rules,
- conformance rules,
- pragmas and names,
- the SYSTEM library unit,
- representation clauses, and
- forcing occurrences.

The maintenance process also includes impact analysis to the Ada Compiler Validation Capability (ACVC) test suite and to the users. Once the interpretation is approved by the AJPO, the ACVC test suite and the ACVC Implementers' Guide are checked to ensure that they conform to the approved interpretation, otherwise, the test suite and/or implementers' guide, need to be revised accordingly. The AJPO's ruling on a commentary does not guarantee immediate updating of the ACVC suite.

Existing Ada software may be affected by language maintenance as new compilers are released and validated to conform to the Ada Standard. The AJPO has issued guidelines defining classes of validated compilers and their use in different phases of a project. Edition 1, Section 3.1, discusses validation policy in depth.

15.4 The Revision of the Ada Standard

The procedures of most standardization bodies include rules governing the revision of a standard. A typical rule requires that the standard be set for at least five years. After that time, the standard may be reviewed. The purpose of this review is to evaluate the standard and to recommend action. Possible recommendations are that the standard be:

- Renewed without change,
- Revised, or
- Withdrawn.

The procedures for a revision follow similar lines as for a new standard. The responsible body announces its intention of revising the standard for the language but, thereafter, the same procedures as for a new standard are followed, involving a public review and discussion period leading to a vote by the appropriate body.

In 1988 the Ada Standard will have existed as a Military Standard and an American National Standard for five years. Currently, there is no official activity underway that would lead to a revised Ada Standard being issued in 1988. It is expected that ANSI will approach the organization that is responsible for the standard, which is the AJPO, and request the status of the standard. The AJPO will probably request that the standard be revised.

Experiences with the revision of other languages have shown that the revision process can last several years, so the revision process will probably be completed in the early 1990's. The goal of the revision process is to make the language more effective. The likelihood of major changes seems small at this point.

The Institute for Defense Analyses (IDA) has been requested to investigate and produce recommendations on how the language should be revised. The person in charge

of this investigation is Dr. John Kramer, who may be reached at:

Dr. John Kramer
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria, VA 22311
(703) 845-2263
AUTOVON 289-1948 ext 2263

Section 16

AFCEA Ada Education Training Study (ADETS)

This section describes the framework and findings of the Armed Forces Communications and Electronics Association (AFCEA) Ada Education and Training Study (ADETS). The report was published in July 1987.

16.1 Purpose of Study

In conjunction with the AJPO, AFCEA organized an Ada education and training study (ADETS) in the fall of 1985. The purpose of the study "was to provide information and recommendations to Department of Defense (DoD), North Atlantic Treaty Organization (NATO), and industry on the education and training required to introduce and support the Ada programming language successfully."²

16.1.1 Study Team Composition: Industry

The ADETS team was composed primarily of members from industry, both software companies and major defense contractors. There were also liaison members with government organizations, specifically the Ada Software Engineering Education and Training (ASEET) team. Members were chosen for their experience in the Ada community, in software development and in education. Progress on the study was briefed to senior officials in the services, the Pentagon, and the AJPO. The report was reviewed by both industry and government executives before its release.

The ADETS team was divided into four subteams: a NATO team plus three U.S. teams of seven members each. These U.S. groups are the Integration Team, the Requirements Team, and the Education and Training Methods Team.

16.1.2 Definition of Effort

The ADETS team was chartered to look at existing training: methods, usage, availability, requirements, and shortfalls. Data were collected and analyzed during a year and a half period. Surveys, case studies, interviews and literature searches were conducted; thus providing a balanced viewpoint of the state of Ada education and training.

The original idea was to develop a model of Ada education and training requirements. By then matching the real world with this model, shortfalls would be identified. In the course of the study, it became apparent that such a model could not be con-

²[ADETS87] Page 1.

structed. There was a lack of firm data on training plans, especially on the numbers of people needing what level of formal exposure to Ada.

16.2 Sources of Data

The data collected during the ADETS came from multiple sources. The report describes both the methods of collecting data and the actual data collected. (See Sections 4.8, 5.1, and Appendices A through E, and H of the ADETS Report for summaries and analyses of this data.)

Data on the status and level of Ada training came from interviews and discussions, questionnaires, and literature. The ADETS team selected four Ada projects as case studies:

- Worldwide Military Command and Control System (WWMCCS) Information System (WIS),
- Maneuver Control System (MCS),
- MILSTAR Communication System, and
- Combat Data Systems (CDS-5).

Both the government program office and the contractor were interviewed concerning the impact of Ada on their project and their approach to Ada training.

The questionnaire, sent to AFCEA member companies and selected program offices, requested organizational data, information on formal training plans, and projections on Ada training needs.

16.3 Principal Findings

The principal findings of the ADETS relate to management commitment and availability of training. The ADETS team formulated four personnel categories from their data analysis and characterized the training requirements for each category. Widely available training in the Ada language was distinguished from training shortfalls in management awareness, software engineering, and support activities. Sections 5 and 6 of the ADETS Report present the data analysis, while Section 7 discusses education and training shortfalls.

16.3.1 Management Commitment

The ADETS Report notes that a much stronger commitment to Ada is needed both from the DoD and from upper management. At the time of researching and writing the ADETS report, the DoD Ada mandate, Directives 3405.1 and 3405.2, had not been issued³. Fear of a new language, lack of available production quality tools, misinformation, mixed signals, and lack of specific plans have all contributed to the perception that top management, both in the government and in industry, had not yet wholeheartedly endorsed Ada software engineering.

16.3.2 Training Shortfalls

The ADETS team found that while language training was more than adequate to meet current demand, there was a lack of demand and supply for non-language, Ada-related courses. Specifically, the team found that training is needed both in management and support personnel awareness and in software engineering. "Awareness" courses too often present an overview of the language instead of a discussion of the relationship between Ada and productivity, reliability, life cycle issues, standardization, and DoD policy [ADETS87].

Software engineering is a vaguely defined term which does not "denote a universally accepted body of principles or procedures."⁴ Various interpretations range from non-programming activities (i.e., design, documentation, etc.) to specific programming practices. The ADETS Report points out not only the need for formalized training in software engineering, including software development practices, but also the insufficient priority placed on this kind of training.

16.3.3 Number of Trained Personnel

In regard to the number of personnel who had received some form of Ada training, the ADETS team reports two interesting findings. More people in industry have received Ada training than available positions on Ada projects. Consequently, the skills acquired in Ada courses are lost over time with the insufficient project workload. With respect to this observation, the ADETS team noted the following caveat: being "trained in Ada" did not reflect the degree of training (superficial exposure through in-depth, hands-on training) received by the data sources. The team felt that familiarization did not constitute adequate training.

³These directives and their implications will be discussed in Edition 4 of the Program Office Guide to Ada.

⁴[ADETS87] Page 7-5.

Secondly, data analysis revealed a perception, on the part of industry, that government personnel lacked Ada language training. Some technical personnel in the program offices were unable to read or understand the Ada designs and software written by contractors.

16.4 Study Recommendations

The ADETS Report outlines recommendations that address not only the issue of Ada training and education but also the broader context in which this issue occurs. The recommendations address DoD policy and planning, management and acquisition/support personnel awareness, and technical issues.

The report stresses the need for a unified policy under active DoD leadership. Such a policy should not only address the use of Ada for software development but should also be reviewed from the perspective of Ada MIL-STDs and hardware development programs. Moreover, the services need to establish training plans for their personnel.

The ADETS Report recommends that DoD develop and offer a top-level Ada awareness course. Program Offices involved in Ada software development, as well as high level officials from the services and DoD, should attend. Furthermore, awareness courses need to be developed for acquisition and support personnel. The team suggested greater participation on the part of universities and the Software Engineering Institute in Ada education and training.

Recommendations on technical issues address the need for mature, production quality tools as well as the need for greater numbers of Ada projects. The report urges that greater emphasis be given to training in software engineering.

Copies of the ADETS Report may be obtained by contacting:

Brig. General, Kirby Lamar, (Ret.)
Director, Corporate Affairs
AFCEA
4400 Fair Lakes Court
Fairfax, VA 22033-3899
(703) 631-6235

Section 17

Program Proving and Verification

Program verification is one of several formal and informal techniques which are meant to increase the user's confidence in the reliability of a program. Program verification specifically refers to formal mathematical proofs of program text. It is a time consuming, intellectually demanding method, for which only limited automation exists. This section explores some of the terminology and issues surrounding program verification. Verification is distinguished from other confidence building techniques, testing in particular. The advantages and disadvantages of program verification are considered.

17.1 Overview

The goal of program verification is to construct a formal proof of one or more properties of a program or program fragment. The objective of verification is to show correctness, in other words, to prove true the assertion that the program behaves in a certain fashion under certain operating conditions. This assertion may range from a complete specification to a limited set of behavior characteristics.

In order to assure the quality of a program, both static and dynamic analyses of the code should be undertaken. Before generating object code, a *compiler* performs a detailed static analysis of the program, making sure that what is written conforms to the language (MIL-STD-1815A in the case of Ada). A *verifier* performs a detailed (static) analysis of the dynamic behavior of the program. In order to meet this goal, a verifier requires that the meaning of the program be completely elucidated. Essentially, this means that all assumptions about the values of variables and their relationships be stated. Thus the original Ada program text must be annotated with information for the verifier. Using program proving jargon, the semantics of the program must be completely specified. Program verification involves:

- assertions about the behavior of the program, for example, values of variables;
- proof rules or axioms that allow the verifier to progress through the assertions.

Assertions are stated using an assertion language such as Anna. Axioms are expressed in a metalanguage that the user of a verifier will not see.

There are several ways of specifying semantics. The following section provides an overview of the terminology used in discussing semantics and constructing proofs.

17.1.1 Program Language Semantics

In order to prove a program, one must first know the semantics of the language in which the program is written. Semantics refers to the meaning of each construct in the programming language, i.e. its behavior. Formal semantics allows the programmer to assign a unique meaning to a program or program fragment. A formal mathematical notation is used to express the semantics. An alternative method treats each computation like a function; verification is then a proof that each function computes its intended result. The notation for the functional model is treated in more depth in [B&N78].

There are three principal methods for formally specifying programming language semantics: denotational, operational, and axiomatic. Different kinds of formal semantics have different purposes. Denotational semantics is useful for unambiguously telling compiler writers how their compilers should work, or for proving that two programs have the same meaning, for example. Operational semantics is useful for unambiguously telling programmers how their programs can be expected to behave—but not very useful for languages like Ada that allow variations among implementations or for any language (including Ada) allowing nondeterminism. Axiomatic semantics is useful for proofs about programs. That is its intended purpose, but it is not the intended purpose of other kinds of semantic definitions.

Axiomatic semantics specifies programming language semantics through a set of proof rules about the behavior of the language's constructs. It is based on reasoning about the relationships between variables and their values at different points in the program text. Using axioms and symbolic logic, it is then possible to prove properties about a program. R. W. Floyd proposed the basic idea, in reasoning about flowcharts. C. A. R. Hoare expanded on this method, reasoning about statements written in a given programming language. Hoare asserted that the ability to reason about statements effectively gave computer scientists a mechanism to specify the semantics of a programming language.

An axiomatic semantics description need not be complete. The proof rules for the language depend on what properties one is trying to prove. The proof rules are based on what subset of the language is being used. For example, if the program contains no tasking constructs, axioms about tasks need not be included. If one is not interested in all the properties of a program, the engineer may be able to use a simpler set of proof rules. To date, axiomatic semantics has shown the greatest promise in the area of program verification and automated verification tools.

The Hoare method develops proof rules or axioms for every statement in a language. The literature discusses the derivation of these rules by class of statement (assignment, conditional, loop, etc.) as well as their combination into proofs about the correctness of a program or program fragment.

17.1.2 Verification Process

The properties of a program, such as the relationship between variables, are formulated as assertions. These assertions, or predicates, are known as preconditions or postconditions, depending on their context in the program text. For any proof rule, the assertions directly preceding it are known as its preconditions, and the assertions immediately following it are called the postconditions. A loop invariant is a property about the loop which is true for each iteration of the loop.

Proofs about programs are generally composed in reverse gear, deriving preconditions from postconditions by applying a set of axioms and the rules of inference. One starts out by assuming that the postconditions are true, and the verifier then tries to show that the preconditions are true at the point of call.

As with the design of a program, a proof is constructed hierarchically. One first verifies a subprogram, then proves a call on that subprogram. Sequential composition may also be used to show that a program fragment is correct. If for some precondition A , statement S_1 is executed and postcondition B is true, and if B is true, statement S_2 is executed and postcondition C is true, then it has been proven that if precondition A is true, postcondition C is true following execution of the sequence S_1, S_2 . This precondition-statement-postcondition can be written as a correctness formula, namely $A\{S_1\}B$. Verification can then be described as the process of deriving valid formulae.

Formally, one starts a proof by giving a precondition for the entire program. Using proof rules, creativity, and mathematical insight, one then derives a postcondition for the entire program. Hopefully, the postcondition reflects what the program should be computing.

The original Floyd/Hoare approach stated that program verification consists of two proofs: partial correctness and termination. A partial correctness proof consists of showing that if a program or program fragment terminates normally, then the assertions made at different points in this fragment hold true. More modern approaches advocate proving termination as an integral part of the proof.

Full program verification requires proof of termination. In other words, a proof must show that a sequence of statements executes to completion. In proving termination, one tries to show that a sequence of statements either terminates normally or terminates with some expected error condition.

The choice of pre- and post-conditions is extremely important. The generality of the pre- and postconditions asserted affects the strength of the reasoning that can be applied. The more general the precondition, the more rigorous is the proof of correctness. The less general the postcondition, the more rigorous the proof is as well. There is no limit to the number of pre- and postconditions that can be asserted.

The formal notation of program verification may at first seem unwieldy; however,

its formalism and lack of ambiguity are necessary to permit a rigorous mathematical proof. Furthermore, the emphasis on precision facilitates the process of automation. Given that all necessary assertions and invariants are stated, the program prover can achieve a high degree of automation.

17.2 Program Testing and Verification

Verification and testing are both means of validating compliance of a computer program. Testing is used to show compliance with requirements; verification shows compliance with formal specifications. There is the danger that the formal specifications may misstate the true requirements. This error will be obvious in a test, but a proof would succeed anyway. Verification involves reasoning about program texts; testing is always based upon observing computations [L&G86]. Verification makes inferences about a set of computations in a program. Some software cannot be easily verified, such as that containing complex tasking features, representation specifications or poor programming practices (such as aliasing—multiple names for the same object). Acceptable operation of such software may be better demonstrated by stress testing.

Tests are written to detect errors as well as to show correct program execution. Myers in [Mye79] defines testing in terms of trying to find fault in a program, not trying to show that the software does not fail. Verification is distinct from testing for several reasons: 1) verification involves a mathematical proof; 2) it proves properties about the program rather than that the program performs as expected; and 3) it is designed to provide a definitive answer, typically a “yes.”

The most common forms of testing are known as white box and black box testing. White box testing tests the logic of the code; it seeks to exercise the different parts of the software. Black box testing, on the other hand, does not take advantage of the information inside the module in the test itself. Black box testing typically tests assumptions about inputs and outputs; for example, can the software handle inputs outside the expected range.

Because verification seeks to prove assertions about statements in a program, it is closer to white box testing than to black box testing. Both white box tests and assertions need the program text in order to be developed.

The software life cycle uses several kinds of testing: unit testing, integration testing, and acceptance testing. These tests focus on the underlying algorithm (unit-level), module interfaces (integration-level) and customer requirements (acceptance-level). Testing is intended to provide traceability to both software design and requirements.

Verification can also be applied at several points in the software life cycle. Individual modules can be verified, similar to unit-testing modules. As the modules are integrated, the verifier can be used to show additional properties of the software. As the size of the

program increases, it becomes harder to verify because of the increasing number and complexity of the assertions.

The use of Ada facilitates unit testing and software integration testing; however, it does not eliminate the need for such tests. An Ada compiler's required interface checking capability automates certain kinds of testing, such as ensuring that different parts of the program communicate through a well-defined "type" interface. In other words, the data types and the direction of data flow are checked for compatibility. Compilers do not test whether or not the data is reasonable, only that it is correctly specified. Ada compilers do insert run-time checks in the executable code. These checks do not substitute for testing. While they do notify the programmer what class of error has occurred (for example, a range constraint violation), it is the programmer's responsibility to determine the cause of the error and to take corrective action.

17.3 Practicality of verification

Program verification is often not undertaken because of its perception as difficult, obscure and academic. However, it should not be dismissed lightly because of the advantages it provides. This section discusses both these benefits and their associated costs. Section 17.4.1 discusses the IBM Cleanroom method which uses verification as part of its methodology. The remainder of Section 17.4 addresses ways of reducing these costs in order to profit from verification techniques.

17.3.1 Benefits

The greatest benefits of program verification are in the areas of program reliability and maintainability. Verification can show that the software will behave in a predictable manner, given a set of input conditions. The ability to prove the correctness of a program or program fragment increases user confidence in that section of program text.

Reliability is a critical property of any software. Reliability serves as an indicator of how well a given program produces usable and predictable results. It may be measured in terms of errors per module, errors per "n" lines of code (e.g., 10 thousand lines), or using more traditional hardware measures such as mean time between failure and average down time. Reliability can also be measured in the context of a system, such as average number of successful hits for some number of launches.

Verification, being a formal proof, attempts to present a "True" or "False" answer. Either the verifier manages to construct the proof or it does not. If it does not, either the assertion one is attempting to prove is false, or it is true but unprovable (a possibility unlikely to arise in practical verification), or it is provable but not by the verifier's theorem prover except perhaps with more advice. If the assertion is false, the program

does not meet its specifications. There are many assertions, or verification conditions, that must be proven true or false. Failure to prove some may (as described above) indicate a program or specification error. Others may be false because loop invariants are too strong or indeterminate (true for some input values, false for others) because loop invariants are too weak.

A proof can enhance the reliability of a program because it guarantees one or more aspects of a program's run-time behavior. For example, if critical algorithms in an embedded application are proven correct, the user or customer has greater confidence in the overall reliability of the system.

Practically speaking, not every aspect of every software module can be verified. Applying verification where possible, however, allows the developer to concentrate testing efforts on those system features which do not lend themselves to verification. According to [Coh86], formal verification should not be confused with validation; formal proofs are one of many methods for validating software, which include also informal proofs, code reviews, and testing.

Verification supports the software quality goal of maintainability in a more indirect manner. An annotated program in and of itself is more complex to read; however, it does force the programmer to state underlying assumptions about the data objects and their values. Thus more information is available to the maintainer regarding the intended behavior of the program text. In the long run, however, an annotated program only serves the interests of maintainability if the proof is maintained as well as the code.

A more subtle benefit of verification is that the exercise of proving a program segment may bring valuable insights to the programmer. The discipline involved in asserting properties about the program's behavior may uncover some subtle errors in the logic of the software. Work done at IBM using the Cleanroom technique [Dye83], discussed in Section 17.4.1, applies this idea.

17.3.2 Difficulties and costs

The Ada language is complex, providing many more constructs than the simple structured programming constructs whose verification is well-documented in the literature. For example, Ada provides arrays, recursion, packages, generics, tasks, overloading, renaming, exceptions, and representation specifications, all powerful features which if judiciously used, permit the creation of reusable, efficient software for an embedded target.

The Ada Language Reference Manual specifies the syntax and semantics of the Ada language. There are parts of the language, however, which are not completely specified: implementation of certain features is allowed to vary. For example, Ada does not require that all uses of uninitialized variables raise the exception `Program.Error`. Compilers

implement pragmas differently: they do not need to support all of the language-defined pragmas, and they may introduce new pragmas as well. Ada allows a nondeterministic order of execution for program unit elaboration, select alternatives within tasks, and conditions connected with the logical operators (and, or, xor).

Program verification is made more difficult because of these variations. A verifier which only handles a specific implementation is by its very nature limited. One which is completely implementation independent, however, cannot prove anything about that part of a program which relies on a specific implementation.

Ideally, verification should be independent of the implementation of the language in which the program is written. In other words, the verifier should not make any assumptions about the code generated by the compiler and other language processing tools. For instance, the verifier should not depend on whether parameters are passed by value or by reference. As another example, the verifier should not depend on whether or not numeric overflow raises the Ada exception `Numeric_Error`. Similarly, the verifier should not assume that the tasking implementation accepts entry calls in the order in which the select alternatives are written, should several entry calls arrive simultaneously.

Implementation independent proof rules are much more difficult to write. At one extreme, such proof rules must account for possibly perverse implementations which raise the exception `Storage_Error` upon startup [CohN86]. In the general case, it becomes extremely expensive, if not impossible, to write a set of proof rules that reflect all possible implementation variations.

The program termination part of verification poses two obstacles: termination is not always a desired property; and termination can be extremely difficult to prove. Certain programs such as operating systems are intended to run forever. Applications such as weather radar and air traffic control systems are designed to scan the skies continuously. Nontermination is a requirement of such software, and termination is effectively an error condition. (Replacing an existing system with a new one requires that at some point the operator terminate the old one; this termination is usually not considered a normal termination.) Even if termination in the traditional sense does not apply (for example a sensor control function), there are still many properties of these programs that can and should be verified.

Modern programming languages, including Ada, often provide programming features which complicate termination proof. Parallelism, loops, and recursion, all supported by Ada, are difficult to verify. In verifying parallelism, one must be able to prove assertions about synchronization and about the lack of deadlock. Ada tasks, moreover, often implement their algorithms with infinite loops. Other loops may execute indefinitely, until some condition is met. (The corresponding Ada constructs are the infinite loop `[loop ... end loop;]` and the while loop `[while condition ... end loop;]`.) It is extremely difficult to prove that either the condition or an error condition occur, re-

sulting in loop termination. Similarly for recursion, it may not be easy to show that a subprogram will call itself only a finite number of times.

Some of the technical obstacles to developing axiomatic proof rules for each kind of Ada statement are being overcome, but more work needs to be done. There are other technical obstacles which are not Ada specific but true of software in general. It is extremely difficult to prove that an entire program is correct. The sheer size of a program accounts for part of this difficulty: because of the many execution paths, their interrelationships, and the large number of interfaces, a large number of assertions must be made.

There is no guarantee that the proof rules themselves or their application are correct. The verifier, if an automated verifier is used, could itself contain bugs. This problem is endemic to using any kind of tool, including such basic ones as editors, compilers, linkers, etc. It should not be construed as a disincentive to use program verification, but it should temper one's confidence that a verified program is guaranteed to be 100% correct.

There are more fundamental problems with regarding a verified program as fail-proof: The compiler or target operating system or even the hardware could contain design errors; the hardware could suffer a transient error; the specification could incorrectly convey the informal requirements; the informal requirements themselves could omit some conditions necessary for what we would intuitively call "correct" processing. Errors in the target operating system are a special case of a more general problem: invoked software that is assumed to be correct actually is not. Formal verification, like testing, should be regarded as no more than a confidence-building measure, though verification should build a much greater degree of confidence than testing.

Program verification is an expensive technique, in terms of labor, machine resources, and time. Verification requires a knowledge of complex logic and mathematics. The notation used in proofs is difficult to read. Therefore, more highly trained individuals are needed to develop assertions and proofs about software. As noted above, there are numerous Ada features which do not lend themselves easily to the commonly used axiomatic verification techniques. Developing predicates that reflect the program's behavior in a meaningful way requires creativity as well as mathematical insight.

In spite of the technical obstacles and expense, verification is a useful technique. The next section addresses ways of making verification less of an academic exercise and more of a practical method in a software project.

17.4 Cost reduction approaches

The high costs and disadvantages of formal methods can be mitigated to achieve some of the benefits of program verification. These approaches involve both general and

Ada-specific software engineering practices. The introduction of formal methods into a software project will reallocate the amount of time spent in different phases of the software life cycle, concentrating greater effort in the early phases, where errors are the least expensive to detect and correct. Integration, acceptance testing, and maintenance will reap the benefits of working with a higher quality, more reliable product.

17.4.1 IBM Cleanroom Technique

The Federal Systems Division of IBM has pioneered a mathematically-based software engineering approach known as the Cleanroom method. The goal of Cleanroom developed software is to create computer programs with a high degree of reliability and a low probability of errors. Formal, structured methods are used for both requirements and design specification. The implementation phase is supported by code inspections, walkthroughs, and formal verification.

Cleanroom development differentiates the test group from the development group. The developers are not allowed to test or debug their own software; they must use the techniques discussed above to ensure that their code is correct before it is released to the test team. The test team is independent and develops tests that reflect the operational environment of the software. Mathematics is also applied to the testing process: the distribution frequency of input data is determined and the test cases executed at any given time are selected randomly.

The University of Maryland has performed an empirical study to assess the Cleanroom technique. The framework of the study is described in [SBB85]. Ten three-person project teams implemented an electronic message system using the Cleanroom method, while five three-person teams did the same project using conventional methods. The data analysis showed that the Cleanroom technique both results in higher quality software and introduces discipline in software development. Key study findings show that Cleanroom software is more correct, meets system requirements, and is less complex than code developed through other means. The separation of development and test teams was found to be very effective in enforcing the use of good software engineering practices and methods. The code produced was more readable and better designed.

Although the Cleanroom case study was applied to a small project in an academic setting, its results show that formal methods can be applied successfully. The combination of formal and informal methods (i.e., verification with code inspections and walkthroughs) preserves the discipline of mathematical proof while allowing the use of alternate, less rigorous methods. Application of a Cleanroom-like technique to a large project in its successive stages of design and development could also help build quality into the software from the beginning. Although it may be unrealistic to deny the use of a compiler to a programming team, incentives should be provided to encourage the pro-

grammers to apply more formal methods in engineering their code. For Ada, compilers ought to be provided with code generation suppressed so that reviewers can exploit the compiler's strong static checks but not use the compiler to conduct unit testing.

17.4.2 Selective Use of Verification

Some of the problems noted in Section 17.3.2 addressed size complexity and difficult features. In [CohN86], several recommendations are discussed which offer a compromise solution to these problems. The fundamental idea is to apply formal methods to software at the component level.

By verifying software at the module level, proofs can be accomplished in a more reasonable amount of time. A library of validated modules (i.e., whose implementation is validated) can be built so that verification of software using these modules is based on their specification, not their implementation.

Instead of trying to prove properties about the entire program, one can apply verification techniques only to the most critical parts of the program. Other less mathematically rigorous validation methods can be applied to other parts of the software. These alternative validation methods include hand proofs, design walkthroughs, code inspections, unit testing, and trustworthy component reuse.

A similar cost reduction approach, frequently used today, is to apply formal verification methods only to the most critical properties of the program, rather than to its most critical algorithms. One such critical property is security.

17.4.3 Verifiable Ada Subset

Implementation variations make it difficult to write proof rules for some of Ada's features. In order to facilitate verification, one can restrict the use of Ada features to the verifiable constructs for certain components. In [Coh86], several such restrictions are suggested. The verifiable subset would not allow the programmer to use aliasing, shared variables or address clauses. The purpose of the verifiable subset is to exclude features which could make proof rules invalid. To use aliasing as an example, if A is used as an alias for B and B is assigned the value 4, one cannot verify that A does not have the value 4.

The restrictions that would be imposed to create a verifiable Ada subset also support good programming practice [Coh86]. In building a large software project, it will be necessary to violate some of these restrictions. The components which must use features outside the subset would then be isolated in a few special-purpose components with well-defined interfaces, consistent with the intent of the Ada language. These modules can be validated through less formal means than mathematical proof. The point is not to

deny the use of unverifiable constructs, but to ensure that they are used in a controlled and meaningful fashion.

Another approach to verification proposed in [CohN86] is the use of “natural semantics.” Natural semantics represents a category of assumptions about an implementation’s behavior in order to write simpler proof rules. The verifier could only be applied to those programs which obeyed these assumptions. The assumptions are principally in the area of optimization, for example, order of evaluation, code motion optimizations, and code generation strategies. Although the verifier would no longer be implementation independent, it would be justifiable if the assumptions were obeyed by a large number of implementations. Recognizing the need for efficiency, Cohen also proposes a special “optimization pragma,” distinct from the predefined Optimize pragma. This new, implementation-defined pragma would allow the compiler to take “unnatural” steps for the sake of efficiency. Modules containing such a pragma would not be verifiable except through other means.

17.5 Summary of current research

Verification assistants are needed to automate the process of finding invariants, stating pre- and postconditions, and deriving the correctness formulae. There are several efforts being funded in this area, described below. Much of this work is still at the research stage and has not resulted in robust products.

17.5.1 Full Semantic Specification of Ada

The Anna language, an acronym for Annotated Ada, is an effort to incorporate assertions about Ada programs into a preprocessable form. Anna, described in [L&H84], introduces two kinds of formal comments to capture the underlying meaning of a program. *Virtual Ada text* is legally correct Ada used to state explicitly underlying assumptions, for example, a **Length** function in a stack package. *Annotations* are used to make true/false assertions and to state axioms about the computations, based on the different language constructs.

Under contract with the European Economic Community, Dansk Datamatik Center has developed a full semantic specification of Ada. An application of this work in building a software development environment is discussed in Section 21. Further information may be obtained from:

Kurt W. Hansen
Dansk Datamatik Center
Lundtoftevej 1C

DK-2800 Lyngby
Denmark
+45 2 87 26 22
khansen@ADA20.ISI.EDU

17.5.2 PolyAnna

Rome Air Development Center is sponsoring an effort to build a prototype verifier based on Anna. (PolyAnna stands for Polymorphic Anna.) The prototype is being built on a Sun workstation, and a version should be available in the fall of 1989. The contractor is Odyssey Research. The point of contact is:

Don Elefante
RADC/COTC
Griffiss Air Force Base, NY 13441
(315) 330-3241
AUTOVON 587-3241
elefante@RADC-MULTICS.ARP

17.5.3 Verifiable Subset Definition

Computational Logic is designing operational semantics for an Ada subset using the Boyer-Moore logic. Their objective is to define a verifiable subset of Ada which lends itself to rigorous proofs. This effort is part of a larger government contract; no point of contact was available at the time of writing. The industry point of contact is:

Michael Smith
Computational Logic
1717 West 6th, Suite 290
Austin, TX 78703
(512) 322-9951

17.5.4 Security Analysis of Ada Programs

Electronic Systems Division (ESD) is sponsoring a two-phase effort. The first phase is studying the security analysis of Ada programs. The second phase, anticipated to begin in early 1988, will develop an automated proof of correctness for Ada programs, using an annotated form of Ada as input to the verifier. The annotation language has not yet been selected. The contractor is CompuSec. The government point of contact is:

Lt John N. Molloy
ESD/SYC-2
14 Oak Heart
MITRE Building L
Bedford, MA 01730
(617) 271-5053

17.5.5 Organizations on Formal Methods

The Special Interest Group on Ada, SIGAda, has a Committee on Formal Methods whose charter is to stimulate the use of formal methods. They are investigating formal verification, formal semantics, and formal specification languages. The chairperson of the Committee is:

Richard A. Platek
Odyssey Research Associates, Incorporated
1283 Trumansburg Road
Ithaca, NY 14850-1313
(607) 277-2020
rplatek@Ada20.ISI.EDU

Both the Naval Research Lab and the Institute for Defense Analyses (IDA) sponsors workshops on Ada Verification. Further information on IDA may be obtained through

Terry Mayfield
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria, VA 22311
(703) 824-5524

Section 18

Environments

This section discusses the evolution of Ada programming environments. The Stoneman definition of an Ada environment, the baseline to which most environments are compared, is reviewed. Several current projects are discussed:

- the Common Ada Programming Support Environment (APSE) Interface Set (CAIS),
- the Software Development and Maintenance Environment (SDME),
- the Software Technology for Adaptable, Reliable Systems (STARS) effort, and
- the Software Life Cycle Support Environment (SLCSE).

Although not all these environments are Stoneman compliant in the strictest sense, they are all characterized by interesting approaches and tools to support program development.

18.1 Stoneman

18.1.1 Definition and Purpose

The Stoneman Document is a recommended standard developed by the Department of Defense. It was written in 1980 for the purpose of defining the requirements of an Ada Programming Support Environment (APSE). It was designed to aid the Ada effort by defining a standard set of development support tools for Ada programmers to use during the life cycle of Ada programs. The Stoneman definition aids portability in two ways. First, it provides guidelines and consistency to the development of Ada environments, so that a development tool meeting Stoneman specifications should be usable in any APSE that also conforms to Stoneman. Second, the document defines a standard set of tools and user interfaces so that programs developed in one Stoneman environment can be transferred (or ported) to another Stoneman environment. Stoneman was intentionally designed to be very general. It was designed to integrate the software tools existing at the time as well as to be flexible enough to incorporate tools that would be developed in the future. The requirements were designed to support Ada applications throughout the software development life cycle, from requirements definition through installation and maintenance.

18.1.2 Structure

Stoneman defines the APSE as being comprised of three major components:

- the database,
- the toolset, and
- the interface.

The information about the Ada development project is stored in the database. It is the central storage area for the entire life cycle of the project. It must be dynamic enough to offer sufficient storage area for all of the tools used during a project's life cycle. It must store objects⁵ and provide facilities to access and modify the objects. These include versions, configurations, history preservation, partitions and access controls. Stoneman requires that the database be as reliable as required for any given project, and that management reports be produced upon request.

The toolset consists of a series of tools that will be used to develop the Ada applications, as well as to perform program management and maintenance. The tools are required to be written in Ada to provide portability. The APSE should always be able to have more tools added to it.

The Stoneman interface requirements include the interface to the user as well as the interface between the operating system, the database, and the toolset. The user interface will permit the user to invoke an individual tool in the toolset.

The Stoneman Document defines an APSE as having two levels, the Minimal Ada Program Support Environment (MAPSE) and the Kernel Ada Program Support Environment (KAPSE). The APSE was defined this way to provide the most portability possible.

The KAPSE provides the interface to the host operating system. It contains the database and provides the communication and run-time support functions. The interface between the KAPSE and both the MAPSE and the APSE is machine independent to promote tool portability. Through this architecture, the tools from the APSE and MAPSE may be ported to another host operating system with only the KAPSE needing to be modified. The KAPSE does not have to be implemented in Ada. It can take advantage of the host machines operating system, filing system, etc.

In the "onion" diagram in Figure 1, the MAPSE sits on top of the KAPSE. The MAPSE is comprised of the minimum set of tools that are necessary and sufficient to develop and maintain Ada programs [Sto80]. In essence the MAPSE is an APSE, and

⁵In the Stoneman Document an object has a name by which it may be uniquely identified in the database. The object has attributes and contains information.

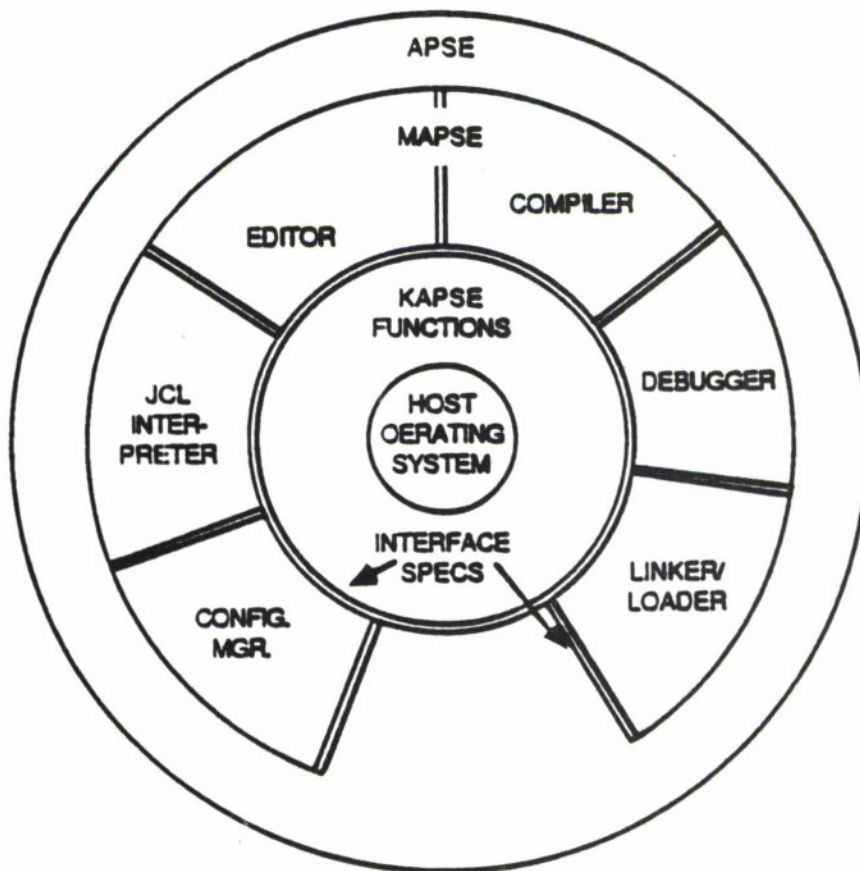


Figure 1: Stoneman APSE Diagram
18-3

it must meet all of the requirements defined for an APSE. Stoneman requires a MAPSE to have the following:

- Text Editor,
- Pretty-printer,
- Translator,
- Linker,
- Loader,
- Set-use Static Analyzer,
- Control Flow Static Analyzer,
- Dynamic Analysis Tool,
- Terminal Interface Routines,
- File Administrator,
- Command Interpreter, and
- Configuration Manager.

A MAPSE just supports basic text manipulation functions, where a more developed APSE might offer a much wider range of facilities. The Stoneman Document suggests that an APSE might support:

- Requirements Specification,
- Overall System Design,
- Program Design,
- Program Verification, and
- Project Management.

18.1.3 Current Use of Stoneman

Originally the DoD funded two development efforts, the Ada Language System (ALS) and the Ada Integrated Environment (AIE), (see Edition 1, Section 6.2). Only the ALS was completed to become a full Stoneman APSE, and it is viewed as an experimental prototype rather than a production environment. The AIE, was redirected to concentrate on the compiler. Many Stoneman requisites require tools to update or check the database, causing the support tools to be slow. For this reason and because of the government's recent push towards non-developmental items (NDI)⁶, Stoneman is viewed more as a guideline than a standard.

In general, the MAPSE is not discussed as being a required part of an environment, since it is actually an APSE. When the minimal requirements for an APSE are discussed, often only a subset of the tools required for the MAPSE in Stoneman are named. The list usually only contains the tools necessary to write, compile and execute an Ada program, such as in the following list:

- Text Editor,
- Translator,
- Linker/Loader,
- Debugger, and
- Compiler.

18.2 CAIS

As described in Edition 1, Section 3.4, of the Program Office Guide to Ada, the Common APSE Interface Set (CAIS) is being designed to permit consistency between the interfaces of the various tools that are to be used within an Ada environment. The CAIS defines the interface between the virtual operating system and the software tools.

The CAIS will be particularly useful for large Ada applications. The set will provide a standard interface for all of the tools in the development environment. Tools developed to conform to CAIS will be able to be ported to any CAIS APSE and still work without code modification. The standard interface will also help with the project database. It will permit movement of components of project databases between APSEs. The interface will provide developers with more choices and more flexibility in selecting the tools for

⁶The NDI push results from the DoD's belief that they are not in the business of developing their own support tools. The belief is that the DoD should make its requirements known and that industry will then meet those requirements through their products.

their project development environment. If changes need to be made in the toolset for a project, there should be fewer restrictions in doing so.

The interface will not encompass all of the possible interfaces to all operating system facilities. It is intended to supply common interfaces to the frequently used system facilities in the widely used operating systems for the purpose of porting tools and project databases to the various environments. CAIS relies on existing standards for interfaces. In some areas, interface standards have not yet been defined or are not yet in a sufficient stage of maturity to be currently included in the CAIS.

18.2.1 Progress on CAIS

The CAIS is being defined in two parts. The first part, known as CAIS DOD-STD-1838, recently became a standard. There are several efforts developing prototypes of the interface. The second phase of the CAIS definition is identified as CAIS Revision A, which upon completion of the standardization process will be identified as DOD-STD-1838A. At this point, there are no known production programs requiring the use of CAIS.

The topics that will be covered in CAIS Revision A include:

- Typing of database objects,
- Distributed environments,
- Standard data interchange format,
- Interprocess synchronization and communication, and
- A improved input, output model.

A functionally complete, operational prototype of CAIS Revision A is also being developed. The prototype delivery date will follow the delivery of the proposed standard CAIS Revision A, in the second half of 1988. CAIS Revision A will probably complete the standardization process in 1989. The government point of contact is:

Patricia Oberndorf
Code 423
Naval Ocean Systems Center
421 Catalina Boulevard
San Diego, CA 92152-5000
(619) 225-6682/7401
AUTOVON 933-6682/7401
oberndor@AJPO.SEI.CMU.EDU

18.2.2 Comparison to Stoneman

The CAIS assumes the general Stoneman definition of the structure of an environment, namely the existence of the KAPSE between the APSE and the operating system. CAIS defines the necessary interfaces for this structure. Stoneman was intentionally designed to be general, to permit flexibility. Now that environments have been created and their interfaces between the various parts differ, it has been determined that an interface standard is necessary to promote portability and consistency. The number and types of tools have increased since Stoneman was written, making it is easier to define the necessary interfaces.

18.2.3 CAIS Implementation Validation

To ensure that CAIS is taken full advantage of, a suite of validation tests is being developed to test Ada environments to measure how well they conform to the standard. The tests will be identified as the CAIS Implementation Validation Capability (CIVC). The philosophy of the CIVC is similar to the ACVC (Ada Compiler Validation Capability), (see Edition 1, Section 2.2.2).

The CIVC will measure the conformance of CAIS implementations to the requirements as defined in the CAIS standards. The CIVC results will aid DoD organizations in choosing a CAIS implementation that is appropriate for their projects.

The first set of CIVC tests will be based on CAIS DOD-STD-1838. They are scheduled to be released in December 1988. More tests will be added to incorporate the interface requirements defined in CAIS Revision A. For more information on CAIS Implementation Validation Capabilities contact:

Ray Szymanski
Air Force Wright Aeronautical Laboratories
AFWAL/AAAF-2
Dayton, OH 45433
(513) 255-2446
AUTOVON 785-2446
rszymanski@ADA20.ISI.EDU

18.3 SDME

The Software Development and Maintenance Environment (SDME) System is being developed for the Worldwide Military Command and Control System (WWMCCS) Information System (WIS). It will support the development and management of WIS

projects. This will include joint mission as well as service and site-unique software.
[SDME86]

The SDME is being developed on a VAX/VMS system. It is designed so that porting to standard WIS equipment, which has not yet been identified, will be possible. The SDME will first be ported to a Honeywell DPS8, the hardware currently in use at WWMCCS sites. There is also a possibility of a future effort to port it to either a IBM/MVS system or a DEC-20 system. On all systems it will operate in one of two modes: a Local Area Network (LAN)-integrated mode, or stand-alone mode.

18.3.1 Structure

The SDME is designed to be portable. It interfaces with a host machine's operating system, compiler and linker. It is a collection of user tools to aid programmers in producing reliable, quality software. There are also tools to aid administrative and managerial personnel in the directing and tracking of WIS projects. The SDME has an on-line help facility to assist users in using the system.

The capabilities of the SDME include program management, configuration management, a COConstructive COst MOdel (COCOMO) tool⁷, and a time line management tool. The SDME will provide users with the general tools such as an editor, a debugger, and a pretty printer. Other features of the SDME will include a tool to perform standards checking on Ada source code and design language files, and a tool to generate quality metrics about either Ada source code or design language units.

Some of the tools are adapted from the set of Naval Ocean Systems Center (NOSC) tools (see Edition 2, Section 14.2). The first phase of the project reviewed the tools in the Ada Repository and determined which would be applicable for the SDME. Then the tools were selected, modified as necessary, integrated, and hosted on the VAX.

At this time the SDME has not been ported to another system, but is scheduled to be rehosted on the Honeywell DPS8 in February 1989. The point of contact for further information is:

Capt James B. Hogan
Headquarters
Electronic Systems Division
ESD/SYW-2P1
Hanscom AFB, MA 01731-5000
(617) 377-4754
AUTOVON 478-4754
hogan@MITRE.ARPA

⁷From the Ada Repository.

18.4 STARS - Environment Products

In order to fulfill the goals of promoting the development of quality, reliable, reusable software, STARS is sponsoring different classes of software projects: applications, environments, and new technology. The STARS Program Management Plan [PMP86] outlines the capabilities it seeks for each of these areas. Sections 19.2 and 21.2 of this edition discuss tools-related efforts. This section focuses on the environment component of the STARS plan.

STARS will sponsor several software engineering environments, both prototype and fully operational versions. These environments will address different problem domains. The fundamental principle driving the environments is a "software-first" approach to system development. Consistent with the trend towards non-developmental items, these environments will be characterized by the use of standard interfaces, the integration of commercial software packages, and the adaptation of reusable components. The software development language will be Ada.

The STARS environment effort embodies some of the Stoneman ideas in the range of functionality required. It differs significantly from Stoneman, however, in the underlying structure of the system. Stoneman describes a layered architecture, whereas the STARS Software Engineering Environment (STARS-SEE), like the SDME, takes advantage of the host operating system and its resident tools (e.g., compilers, file system manager, editors, etc.). The STARS-SEE description in [TPP86] lists "invariant principles" with which the environment must comply. These principles create a framework in which:

- Ada plays a key role, both as implementation language and basis for command language,
- commercial software must be used wherever possible,
- support for reusability, configuration management, and program management must be provided,
- communications with geographically distributed systems must be supported,
- standard interfaces must be used for graphics, databases, networks, etc., and
- system overhead and response time must be reasonable.

The STARS environment procurement is underway. Current plans call for fielding three prototype environments by the end of government fiscal year 1988 and three production quality environments by the end of fiscal 1991. The prototypes will be subject to peer review and evaluation [TPP86]. Further information on STARS is available through:

Col Joseph S. Greene, Jr.
Director, STARS
STARS JPO
Office of Secretary of Defense
OUSDRE (R&AT/CET), The Pentagon, Rm. 3E114
Washington, DC 20301
(202) 694-0210

18.5 Software Life Cycle Support Environment

The Software Life Cycle Support Environment (SLCSE) is designed to support Mission Critical Computer System (MCCS) software. The SLCSE provides a user interface and project database which act as the framework for the system, with which an almost unlimited number of software tools may be integrated. It is designed to be methodology independent so that tools that support different software engineering methods may be used with the system. This would give a user greater flexibility in choosing software, such as off-the-shelf tools or those tools designed for the SLCSE.

Besides supporting software development in Ada, it also supports FORTRAN, JOVIAL J73, COBOL, and PROLOG. The host configuration for the SLCSE consists of several DEC VAX/VMS⁸ computer systems, DEC VAXStation II⁹ workstations, and a Britton-Lee Intelligent Database Machine. A delivery of an advanced development prototype is due in August 1988.

The user interface is a menu-driven, multi-windowed interface which must be used to interface with the various tools. There will also be several on-line facilities to assist users. These on-line facilities include a help facility, an advice facility, and a training facility. The database is the central repository for the data collected from the use of the various tools. The collection and management of the data will be performed automatically. The system will be able to generate automatically DoD standard documentation and specifications. The system will be able to estimate the effect of a change to the requirements, design, or code of the developed software.

The point of contact for further information pertaining to the Software Life Cycle Support Environment is:

Frank S. LaMonica
RADC/COEE
Griffiss Air Force Base, N.Y. 13441
(315) 330-2054

⁸VAX/VMS is a trademark of the Digital Equipment Corporation.

⁹VAXStation is a trademark of the Digital Equipment Corporation.

AUTOVON 587-2054
lamonica@RADC-SOFTVAX.ARPA

18.5.1 SLCSE Tools

Two of the tools under development for the Software Life Cycle Support Environment are the Automated Measurement System (AMS) and the Ada Test and Verification System (ATVS).

The Automated Measurement System (AMS) provides tracking capabilities of resources used and the quality of the software being developed. Development estimates are entered into the system and then updated through the life cycle of the project. The AMS provides reports on how the project is progressing in respect to the goals input in the system by the manager.

The AMS collects data manually and automatically. A forms manager provides a format for manual data collection. Currently automatic data collection for the requirements and design phases is limited to two off-the-shelf tools, the Requirements Specification Language/Requirements Engineering Validation System (RSL/REVS) and the Software Design and Documentation Language (SDDL). For both the Ada and FORTRAN development cycles, there is automatic data collection.

The AMS analyzes the collected data, and through the AMS Report Generator, shares the information with the user. The project goals that are not being met are identified on bar graphs, permitting possible quality problems to be identified.

The AMS was delivered in February 1987 on a DEC VAX/VMS system. It is written in FORTRAN-77. The point of contact for further information is:

Roger J. Dziegiel, Jr.
RADC/COEE
Griffiss Air Force Base, NY 13441
(315) 330-2054
AUTOVON 587-2054
dziegiel@RADC-SOFTVAX.ARPA

The second tool that will be part of the SLCSE is the Ada Test and Verification System (ATVS). It can be used during coding, testing and maintenance. The ATVS checks the source code to make sure it is in accordance with the MIL-STD-1815A. When this is verified it can perform static and dynamic analyses on the code. Static analysis can be performed on the program without executing it, unlike dynamic analysis which requires the program to be executed. Besides the error checking provided by a compiler, the ATVS can also perform:

- Program standards checking,
- Software quality measurement data collection, and
- the Generation of reports.

In addition to being a component of the SLCSE, the Ada Test and Verification System will also have a stand-alone version. It is scheduled to be delivered in August 1988. For further information contact:

Deborah A. Cerino
RADC/COEE
Griffiss Air Force Base, NY 13441
(315) 330-2054
AUTOVON 587-2054
cerino@RADC-SOFTVAX.ARPA

Section 19

Tools

Many tools have been discussed in the previous two editions of the Program Office Guide to Ada.

More tools are being developed all the time, some as parts of other tools or environments and others as stand-alone tools. When the language effort was just beginning, major tool development emphasis was on compilers, linkers, debuggers, etc. Now as Ada is becoming established and the use of Ada is increasing, more diverse tools are being developed. At the moment, many of the more sophisticated tools are in prototype form.

Research is ongoing in tools to aid in the development of affordable and reliable software. Tools that verify software against project coding standards are being investigated. Tools that generate code from various design methodologies are becoming available (see Edition 2, Section 8.4). Since Ada was designed to be portable and, hence, reusable on various systems, it is no surprise that there are several ongoing projects to develop reusable software component libraries [Tra87]. Two existing commercial libraries and a public-domain library were discussed in Edition 2, Section 14. Recently, interest has developed in targeting many existing computer-aided software engineering tools to Ada, as will be discussed in Section 21.

The following sections describe two prototypes. The first is the Interactive Ada Workstation which is designed to improve programmer productivity in the development of Ada code. The second section addresses the scope of the STARS effort. The third was an effort to automate the process of developing software, from high level design to a template for Ada code.

19.1 Interactive Ada Workstation

19.1.1 Description and Purpose

The Interactive Ada Workstation (IAW) is a prototype of a software engineering workstation. The system is designed to improve programmer productivity to help meet the DoD's need for less expensive mission critical software. The system supplies tools that permit the programmer to spend more energy on correct design development and less on worrying about syntactic errors. This is accomplished by providing the user with editors that can be used to represent the design in graphical or tabular form. The designs are verified, and then Ada code is automatically generated.

The IAW is being developed in a series of seven prototypes. Different parts of the IAW are developed with each prototype. The prototypes are then tested and suggestions

for improvement are made.

The IAW encourages the use of rapid prototyping as a software development technique [Mar87]. The programmer can create working prototypes of sections of the software throughout the development effort but, in particular, early in the project. This permits testing and interaction with the users. Prototypes of user interfaces which a user can test and evaluate are important because it is often difficult for a user to visualize and describe his needs. Moreover, a user will recognize what he does not like or need. User requirements often change, and a development system which supports rapid prototyping allows both user and implementer to assess the success and impact of such changes.

19.1.2 Structure

The IAW can be broken down into three main groups:

- Host environment support,
- Program development tools, and
- Program support tools.

The host environment for the prototypes is a Symbolics 3600 LISP machine. This system was chosen to take advantage of the LISP tools in providing the Ada Programming Support Environment (APSE).

The tools for program development will include several editors that support various design methods: graphical, spread-sheet, and syntax directed. These editors will generate Ada code. The generated code can be tested and presented to the user to determine if the design is meeting the project requirements. This method will allow the programmer to test the design early in the project and to go through numerous iterations of the design process.

Program development tools that will be included in the IAW are listed below.

- BRAT Diagram Editor:

The BRAT Diagram Editor is based on the design methodology developed by Dr. R.J.A. Buhr. This editor is used for doing high level system design.

- State Diagram Editor:

With the State Diagram Editor the user can graphically represent states of the program software and transitions between the states. The diagram can be tested by defining the conditions which cause transitions between states, input values for the conditions, and expected output values.

- **Decision Table Editor:**

The Decision Table Editor allows the user to represent each state in a table, with a set of conditions, rules and actions. As with the State Diagram Editor, the design can then be tested.

- **IAda Editor:**

The IAda Editor is a syntax-directed editor which will allow the user to write a program without using one of the other editors. It can also be used to modify the output of the other editors.

- **IAda Interpreter/Debugger:**

The IAda Interpreter/Debugger provides the user with a way to execute and debug the IAda code generated by the various editors. The program can be stepped through, and the program variables can be examined.

The Program Support tools planned for the IAW are listed below.

- **Entity Manager:**

The Entity Manager is responsible for the organization and maintenance of the project database. The user will be able to retrieve information from the database from any point in an IAW session [Mar87].

- **Smart Librarian:**

The Smart Librarian task is solely a research effort, whose goal is to use expert system technology to identify reusable software components based on requirements provided by the user. There will be several versions of limited scope of the prototype Smart Librarian developed.

- **Smart Help System:**

The Smart Help System will be an on-line help facility, with the goal of eliminating the need for users to search through manuals. Using expert system technology, the system will help the user choose the appropriate command or sequence of commands.

19.1.3 Current Status

At the time of writing, the third of seven prototypes of the IAW was completed in May 1987. Delivered with the third prototype were the BRAT Diagram Editor, the State Diagram Editor, the Decision Table Editor and the IAda Editor. The IAda Editor is working for a 40% subset of Ada.

The IAda Interpreter/Debugger is planned to be delivered for the fourth prototype, for the current 40% subset of Ada. The first version of the Smart Librarian is also scheduled for the fourth prototype.

19.1.4 Comparison to Stoneman

The IAW contains the three major parts of an APSE: a database, a toolset and an interface, as defined in the Stoneman document. However, it does not conform to all of the other requirements. There is a strong reliance on the LISP-oriented environment on the Symbolics 3600, primarily because the tools are written in LISP. For this reason the IAW is not readily portable. However, future plans include converting the IAW code to the "C" programming language and re-hosting the system to more widely used systems, such as the VAX, Sun, Apollo, etc.

The IAW, also, does not provide a compiler. It is meant to aid programmers in developing Ada code which then would be compiled elsewhere. In lieu of the compiler, the IAW provides an interpreter which performs syntactic and semantic checks on the IAda code. The IAda code is executed by the interpreter, allowing the user to see the effects of changes dynamically. In a compiler based system, the user would have to recompile, relink, and re-execute the source program.

The IAW provides powerful design tools to facilitate and automate the development of Ada software. Some applications may ultimately need to be run on a machine other than the IAW host. The user can port the IAda source code to another host which would provide tools for compilation and execution. (This step could be done through communications software or magnetic tape transfer.)

The point of contact for further information is:

Lt Robert Marmelstein
AFWAL/AAAF
Air Force Wright Aeronautical Labs
Wright-Patterson Air Force Base, OH 45433-6543
(513) 255-6548/3947
AUTOVON 785-6548/3947
rmarmelstein@ADA20.ISI.EDU

19.2 STARS - Technology Development

Technology development is part of the STARS research and development activities. The STARS plan calls for developing reusable Ada components to support a software-first approach [PMP86]. These components will be designed with the objectives of

Computer-Aided Software Engineering systems in mind (see Section 21.2). The tools will be made widely available in a software repository. Initially the tools will be in the Ada repository on the Simtel-20. (See Edition 2, Section 14.2.) Software capabilities are described in terms of a common software base or foundation. The foundation areas identified in which to develop Ada building-block components include:

- Ada as a common command language,
- Software Design, description and analysis tools,
- Text processing,
- Database management systems,
- Operating systems,
- Planning and optimization algorithms,
- Graphics support,
- Telecommunication and network protocols, and
- Other components that comply with STARS program objectives for virtual interface support and machine independent applications.

STARS and the Naval Research Laboratory have awarded 32 contracts to develop Ada programs in these “foundation” areas [FCW87]. See Sections 18.4 and 20.2.3 for further information on the STARS program.

19.3 Requirements to PDL Tool

This project investigated a reliable technique for performing the transition from high level design activities to detailed design and coding activities. The effort focused on creating a prototype to generate automatically Ada Program Design Language (PDL) from a high level design expressed in IDEF₀, the Air Force’s Integrated Computer-Aided Manufacturing (ICAM) Definition Language. IDEF₀ is a subset of SADT¹⁰ (Structured Analysis and Design Technique), is a graphical engineering methodology developed specifically for defining requirements and design. The transformation implemented a method for mapping an IDEF₀ description into Ada PDL, developed during a Reusability Study for the Army Information Systems Engineering Command (ISEC).

The IDEF₀ design method provides a highly modular system decomposition. An IDEF₀ model contains two kinds of information; activities, and the data relationships

¹⁰SADT is a registered trademark of SofTech, Inc.

between the activities. The transformation method uses explicit information in the syntax of the IDEF₀ diagrams to create the initial Ada PDL package specifications. The more implicit information in the semantics of the IDEF₀ diagrams is used to refine these specifications and to produce the Ada package bodies. Figure 2 shows an IDEF₀ diagram and the corresponding PDL generated by the prototype.

A prototype tool was developed by MITRE [M&C??], during the SAda project, that supports user creation of SADT diagrams on an IBM PC. This tool prototype was used as the front end for the automation process of generating PDL from an IDEF₀ diagram. Enhancements were made to generate Ada PDL from the diagram drawn on the MITRE prototype [HVR??]. This new prototype system produces generic templates of Ada code based on the syntactic content of the source diagrams. Refining these templates, based upon the relationships implicit in the diagram, has been left for future efforts.

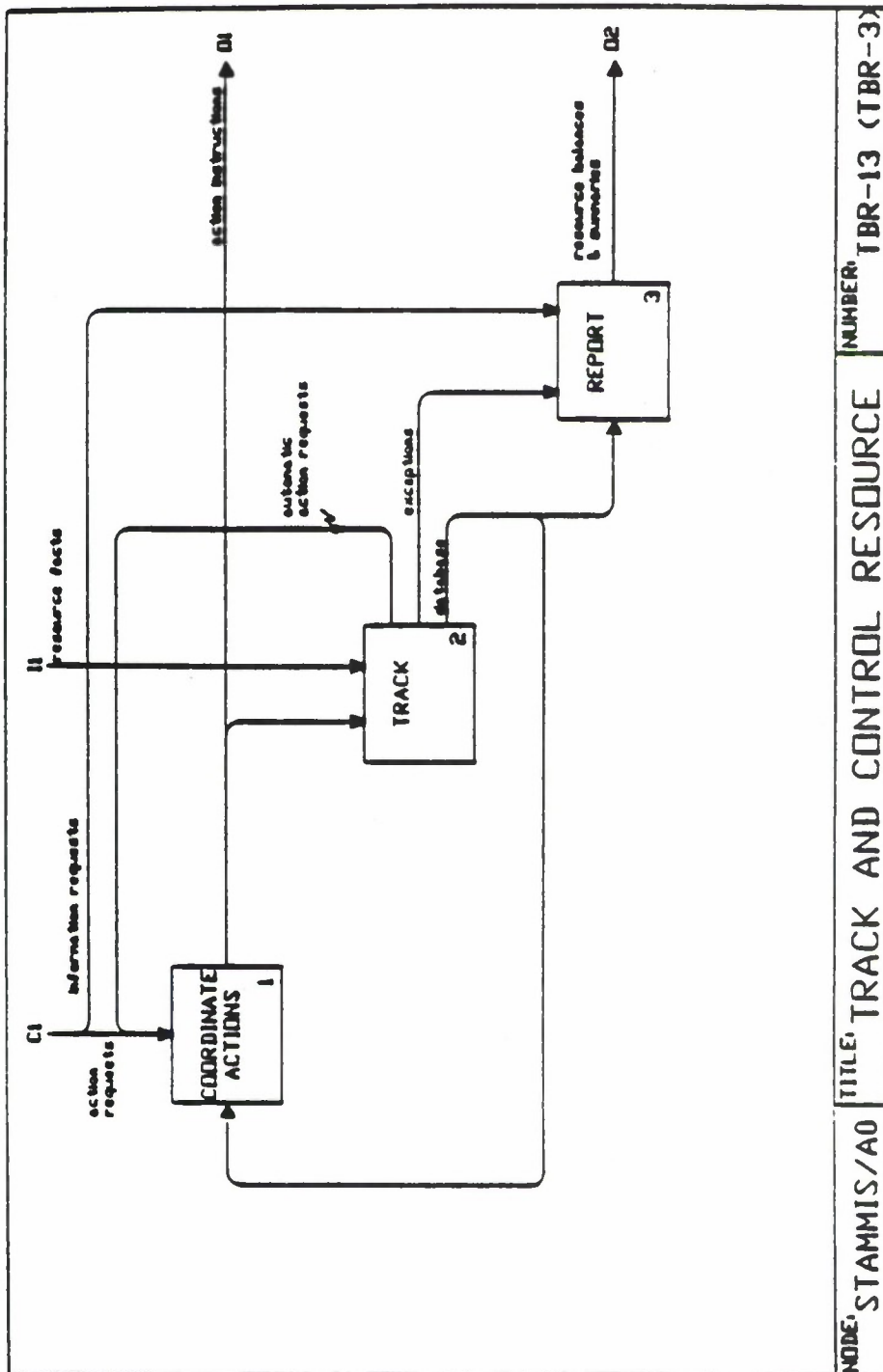


Figure 2: Top Level IDEF0 Design Model
19-7

```

WITH tbd;
WITH COORDINATE_ACTIONS;
WITH TRACK;
WITH REPORT;
PACKAGE TRACK_AND_CONTROL_RESOURCE IS
    PROCEDURE Set_information_requests(information_requests:
        information_requests_Type);
    PROCEDURE Get_resource_facts(resource_facts: resource_facts_Type);
    TYPE database_Com is private;
    PROCEDURE Set_action_requests(action_requests:
        action_requests_Type);
PRIVATE
    TYPE database_Com is
        record
            database: database_Type;
            database: database_Type;
        end record;
END TRACK_AND_CONTROL_RESOURCE;

generic
    type action_requests_Type is private;
    type automatic_action_requests_Type is private;
package COORDINATE_ACTIONS_Template is
    PROCEDURE Get_database(database: database_Type);
    PROCEDURE Set_action_requests(actions_requests:
        action_requests_Type);
    PROCEDURE Set_automatic_action_requests
        (automatic_action_requests:
            automatic_action_requests_Type);
end COORDINATE_ACTIONS_Template;

generic
    type Unknown_type is private;
package TRACK_Template is
    PROCEDURE Get_resource_facts(resource_facts: resource_facts_Type);
    PROCEDURE Set_Unknown(Unknown: Unknown_Type);
end TRACK_Template;

generic
    type information_requests_Type is private;
    type exceptions_Type is private;
package REPORT_Template is
    PROCEDURE Set_information_requests(information_requests:
        information_requests_Type);
    PROCEDURE Get_Unknown(Unknown: Unknown_Type);
    PROCEDURE Set_exceptions(exceptions: exceptions_Type);
end REPORT_Template;

```

Figure 3: Prototype Translation of A0 into Package Specification

Section 20

Interfaces

This section describes some of the issues and efforts involved in interfacing Ada with non-Ada software. Ada's overall philosophy and approach to interfaces are discussed. Specific examples are given for commercially available software packages such as Database Management Systems (DBMSs) and Fourth Generation Languages (4GLs).

20.1 Ada Approach to Interfaces

This section discusses Ada's philosophy towards interfacing Ada software with non-Ada software. Section 20.1.1 describes the history and rationale for Ada's philosophy and 20.1.2 describes current interface approaches being taken.

20.1.1 History and Rationale

Historically, Ada was intended to be used in real-time embedded military applications. The intent was for the entire software system to be written in Ada with some minimal interfacing between the application software and the hardware or some small, special-purpose, non-Ada software.

For the purpose of interfacing Ada applications to non-Ada software, the language provides distinct features such as **pragma** Interface and code procedures. **Pragma** Interface allows an Ada programmer to call non-Ada program modules as if they were Ada subprograms. Code procedures are Ada procedures which contain actual machine code. Along with these features, Ada specifies semantic rules governing their use. These rules are intended to preserve the spirit of the Ada language by supporting encapsulation, modularity, information hiding, and portability.

In theory, using these language features, Ada can interface with any non-Ada software. The difficulties arise in two areas:

- these language features are compiler dependent; the compiler may choose the languages it supports for interfacing, and
- Ada does not address interfacing with entire subsystems directly; using **pragma** Interface for each subsystem provided operation will prove unwieldy and disorganized for the programmer.

Recently Ada is being used in non-embedded military applications such as information systems. Through use in these applications, we are recognizing a need to interface

with Commercial Off-The-Shelf (COTS) software such as graphic packages, database management systems, and knowledge-based expert systems. This is a major area of research for the DoD. The Ada Board has created a new panel which will study Ada Technology and Standards. The chairperson is:

Bill Carlson
Intermetrics, Inc.
4733 Bethesda Ave.
Suite 415
Bethesda, MD 20814

Part of this panel's efforts will be directed at studying Ada bindings to other languages.

20.1.2 Proliferation vs. Standardization

A major theme that has applied throughout the Ada effort is the emphasis of defining and adhering to standards. This theme applies as well to the area of interfacing Ada to non-Ada software. The goal is to avoid the development of several different interface approaches, all of which work and have merit. Having several different approaches severely limits the portability and independence of the software that is developed using a non-standard interface approach. The Ada software will only work with non-Ada software that interfaces using the same approach. Therefore, in order to change, upgrade or replace the non-Ada software, the Ada software must be modified as well. Using a standard interface avoids this problem.

Current research in the area of interfacing Ada with non-Ada software systems stresses the use of existing standards upon which Ada language interfaces, or bindings, are developed. The emphasis here is in adopting an existing industry standard, if one exists, for the standard Ada interface rather than developing a new interface standard just for the Ada language. The advantages of this approach are:

- Programmers who have written non-Ada software which interfaces to software packages using the standard interface will not have to learn a new interface.
- The Ada application software will be independent of the non-Ada software package so long as it adheres to the standard interface. The non-Ada software package can be replaced with another software package that performs the same operations and supports the standard interface without modification of the Ada software.
- Vendors who already support the industry standard for other languages will expend less effort to support the interface standard for Ada.

20.2 DBMSs and Ada

This section describes some of the issues involved in interfacing Ada with COTS DBMSs. Subsection 20.2.1 describes the interface approaches and issues involved. Subsection 20.2.2 contains specifics on the Ada/SQL (Structured Query Language) Proposed Binding. Section 20.2.3 lists some ongoing efforts in defining an Ada/DBMS interface.

20.2.1 Description and Purpose

The goals of interfacing Ada with a DBMS are similar to the goals of the Ada effort itself — to reduce implementation and maintenance costs through the use of standards, and to encourage maintainability, portability, and reusability. By interfacing through a standard interface, the Ada programs will be independent of the underlying DBMS which actually provides database support.

The issues of interfacing Ada to DBMSs are currently receiving attention. There are several approaches that are being considered. In the three approaches mentioned here, SQL is the basis for the interface. SQL is well known, well supported by DBMS vendors, well documented, and is also an ANSI standard.

The first approach is the *preprocessor approach*. This approach is currently in use in industry. In this approach, the programmer embeds SQL statements in the application program code. Prior to compiling the application, it is submitted to a preprocessor which translates the SQL statements into source code which will interface directly with the DBMS.

Although this approach does not meet with favor within the Ada community because the source code is no longer pure Ada, it is the approach that is currently supported by at least three DBMS vendors. These are Oracle Corporation's Oracle¹¹ DBMS, Relational Technologies' Ingres¹² DBMS, and Software AG's ADABAS DBMS. Each of these DBMSs supports an SQL or SQL-like interface to Ada.

In the second approach, which finds more favor in the Ada community, the SQL statements are Ada procedure calls. They follow the SQL syntax as closely as the Ada syntax will allow. The Ada/SQL procedures are contained in an Ada package, and the actual interface to the DBMS is contained in the package body, thus supporting encapsulation and information hiding. This is the approach taken by the Ada/SQL Proposed Binding.

A third approach is referred to as the module approach. This approach would keep the SQL and Ada code fairly separate. All of the SQL calls would be in SQL modules, and compiled separately. The Ada programs would use the pragma interface to call the

¹¹Oracle is a registered trademark of the Oracle Corporation.

¹²Ingres is a trademark of Relational Technology, Inc.

SQL routines. If there were changes made to the database, it is possible that the Ada application would not have to be recompiled.

20.2.2 Ada/SQL Proposed Binding

The proposed binding has two parts: the Data Definition Language (DDL) and the Data Manipulation Language (DML). The DDL describes the data in a form that both the application program and the DBMS can accept. The DDL:

- is standard Ada,
- may be translated into the DDL used by any underlying DBMS, and
- contains augmented information that can be used to generate test data automatically.

The DML are commands such as `select`, `fetch`, `open`, etc. Although standard Ada, the DML is also similar to SQL syntax. This provides all the power and flexibility of both the Ada language and SQL. Major portions of the system described in the paper *ADA/SQL: A Standard, Portable Ada DBMS Interface*, [F&B86] have been implemented on a prototype basis to prove the feasibility of the approach. An Ada/SQL Working Group was formed in early 1987 to participate in reviewing the proposed binding and in discussing technical issues. The chairperson of this group is:

Bill Brykczynski
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria, VA 22311
brykczyn@AJPO.SEL.CMU.EDU

20.2.3 Current Ada/DBMS Efforts

The following companies were listed in Federal Computer Week [FCW87] as those companies the DoD has chosen to do DBMS work for Ada. The funding source for these efforts is the DoD's Software Technology for Adaptable Reliable Systems (STARS) program and the Naval Research Lab.

- Stanford Research Institute, Menlo Park, Ca., for a DBMS supported information management system.

- Grumman Data Systems, Woodbury, N.Y., for an Ada/SQL test generator DBMS.
- Computer Sciences Corporation, Falls Church, Va., for transparent sequential input/output support.
- SAIC Comsystems, McLean, Va., for an Ada report production system.
- Unisys Corp., Paoli, Pa., for reusability library framework.

The WIS (World Wide Military Command and Control System (WWMCCS) Information System) effort has been heavily involved in the Ada/DBMS interface effort. WIS provided the original support for the development of the Ada/SQL Proposed Binding. They have also funded Computer Corporation of America, Cambridge, MA to implement this binding for their DBMS Model 204. Further information may be obtained from:

Lt Col Terry Courtwright
 WIS JPMO
 Washington, D.C. 20330
 (703) 285-5067
 court@MITRE.ARPA

20.3 4GLs and Ada

There two ways in which Ada and fourth generation languages (4GLs) interact. The first way is in using 4GLs, such as program generators, during the production of Ada programs. This is discussed in Section 20.3.1. The second way is in interfacing Ada applications to expert systems. This was discussed in Edition 1, Section 7.2.

20.3.1 Program Generators

4GLs such as program generators are widely used in industry and DoD for languages such as FORTRAN and COBOL. Demand for such tools for use in developing Ada programs is increasing. This raises some issues that are currently being researched:

- A program generated by a 4GL may not take full advantage of the features of the Ada language,
- The 4GL may produce unmaintainable code, and
- The 4GL may produce slow or inefficient code.

The above points will be especially true of programs generated by translating a program written in some other programming language. In general the quality of the Ada code produced will depend on the quality and intelligence of the 4GL tool.

The advantage of these tools is that less development time will be necessary and there will be less development cost. Industry sentiment tends to favor the development of these tools for Ada so long as they produce *maintainable* Ada code.

Some examples of efforts being made in this area are:

- In the area of network protocol prototype, WIS is supporting the specification of a tool which will automatically generate Ada protocol software. [ABB86]
- Research from Bradford University, England includes a dialogue development system for the Ada programming language which supports the production of multi-level adaptable interfaces. [B&R86] The system, ADDS (Ada Dialogue Development System), is designed for and implemented in Ada. In addition, development tools are provided that enable dialogue specific software to be automatically generated from the constituent specifications.

20.3.2 Artificial Intelligence and Ada

Due to projects such as the Strategic Defense Initiative (SDI), the DoD is recognizing the need to use artificial intelligence (AI) in their applications. Current literature and industry belief range from "Ada is better than LISP for AI applications" to "Ada cannot cut it in the AI arena."

It is argued that any language can be used for AI applications. The issue is not necessarily "Can Ada be used?" so much as the issue "Is using Ada more cost effective?" Questions such as, "If it takes twice the time and cost to implement, but is far easier to maintain and debug, then is Ada better?" are being asked. Edition 1, Section 7.2 explored these topics.

There are several important technical issues to be evaluated when deciding whether Ada can handle a particular AI task. Some of these are: representation of parallelism, object oriented programming, and program verification and validation.

Massive parallelism is seen as a way to achieve the type of computational power required for many advanced forms of artificial intelligence. Ada with its tasking feature has a built-in method for specifying parallel operations, a needed capability. Active research is investigating very promising alternate methods for representing parallelism. In the future, it may be advantageous to adopt one of these alternate models for the construction of new AI software.

A central concept in today's AI world is object oriented programming. This ap-

proach is best known in terms of Smalltalk and LISP Flavors. There are three major components to an object oriented programming language: data abstraction, inheritance¹³, and program elements as objects. Ada supports data abstraction very well, but its support of inheritance and program elements as objects is awkward. Ada provides only a limited capability in order to strike a balance between the power of these concepts and their ability to introduce large numbers of errors into an application. Ada provides the capability to pass operations and data onto dependent or "child" packages, but the language requires the developer to specify exactly what is to be passed and how it is to be used. This is extra effort but avoids many errors.

The last issue is validation and verification of an AI system once it is built. This is the process of ensuring that, in a military environment where Expert Systems are used, the software does its intended job correctly. This task is not easy. Verification of programs written in Ada is very complex (see Edition 3, Section 17). The typical AI languages such as Smalltalk or LISP with Flavors are far more difficult. These languages are not strongly typed; in fact LISP allows dynamic binding of symbols to objects of varying type. The scope of symbol binding in LISP is not limited by lexical scope. Procedures can be passed and modified as data before execution. Industry belief is that if systems written in Ada cannot be verified and validated, then systems written in a language such as LISP are even less likely to be verified and validated.

AI applications span many areas including robotics, natural language, machine vision, speech synthesis, knowledge representation, and expert systems. Efforts, to date, are mainly underway in the area of interfacing Ada with AI Expert Systems. A synopsis of these efforts follows.

Sirius, Inc. has developed ALICE (Ada/Lattice Integrated Conceptual Environment). ALICE is a set of tools written in Ada to extend and annotate Ada for AI.

The Knowledge-Based Maintenance System (KNOMES) prototype [BRF86] for the Remote Manipulator System (RMS) of the NASA Space Station Mobile Service Center was built from a software architecture based on Ada tasking and packaging. "Each module in the system contains Ada packages of standard systems services, which interface with an artificial intelligence knowledge-based system (AI/KBS) language component that performs knowledge-based reasoning." By using Ada as the fundamental structure, a well-structured, maintainable program was produced; by retaining the AI/KBS language component, a system able to capture the knowledge needed to solve ill-structured, dynamic, and/or non-algorithmic problems was produced.

¹³Inheritance describes the ability of passing procedures and operations on a given object to new data objects derived from the first one. A new data object may inherit from one or more parents depending on the language.

20.4 Graphics and Ada

In the area of interfacing Ada to graphic systems, the same software engineering issues apply. The interface standard being adopted for Ada is the Graphical Kernel System (GKS), an ANSI standard. This standard is currently used in industry and supports applications written in other languages such as FORTRAN. Commercial and research efforts in the Ada/GKS interface area are underway. For example, Software Technology Inc., Melbourne, Fla., is devising an Ada binding to GKS [FCW87]. In beta tests, this binding has already proved workable in Ada and is reported to execute almost as fast as FORTRAN.

A Master's thesis at the Air Force Institute of Technology [Han86] reports that "an abstract interface with a graphical environment was accomplished." The system contains two graphics programming environments. The first used **pragma** Interface to achieve access to a FORTRAN library of GKS commands. The second was implemented through calls to the first and focused on creating and displaying graphics data.

The WIS effort is also supporting the development of the WIS Graphics Systems. [FHD86] This provides an interface which will enhance the productivity of both developers and users of applications operating within the WIS environment. The specifications for this system were developed by experts in the computer graphics field.

20.5 Other efforts

According to [FCW87], the DoD also expects to fund work on developing an interface with X Windows, as proposed by Science Applications International Corp., San Diego, CA. X Windows technology was developed by researchers at the Massachusetts Institute of Technology and is software available in the public domain. An interface to X Windows for Ada aids in the development of state-of-the-art workstations for Ada engineers.

Section 21

CASE (Computer-Aided Software Engineering)

Today, a new software technology is beginning to emerge, based on the personal workstation equipped with Computer-Aided Software Engineering (CASE) tools (often used on a network with a mainframe providing the needed coordination between workstations). It provides a combination of software tools and methodologies that facilitate an automated discipline for software development, maintenance, and project management.

21.1 Description and Purpose

Using a computer to aid software development has been a goal of researchers almost since the computer was invented. Many substantial advances in productivity have resulted from application of computer resources to specific aspects of software development (e.g., compilers). Since the early to mid 1970s some researchers have worked on the concept of an integrated set of computer tools supporting the entire software life cycle. Some of these early efforts focused on the development of real-time embedded systems (TRW - SREM, DCDS; Teledyne Brown - TAGS), but the majority focused on the management information system world (Yourdon, etc.).

In the mid to late 1980s the expanding power and lower price of personal workstation hardware brought a new popularity to these ideas. There was a simultaneous change in the integrated methods themselves. These changes were caused by a combination of more sophisticated host machines (i.e., powerful workstations), and new application areas that combined the properties of real-time and information management applications.

The CASE tools are different from other software tools in that they focus on the productivity of the individual, professional software developer. They are designed to:

- support a dedicated, personal computing environment,
- use graphics to specify and document software systems,
- link together all phases of the software life cycle,
- capture on the computer all information about an evolving software system from initial requirements through ongoing maintenance activities, and
- use artificial intelligence to perform many routine tasks automatically.

A CASE System is a set of integrated CASE tools that share a common user interface and run in a common computer environment. CASE systems provide computerized

assistance for the development, maintenance, and management of software systems. They differ from earlier programming environments in their breadth of coverage of the software life cycle.

The objectives of a CASE system are:

- greater overall control of software development and maintenance,
- improved productivity, reduced software costs, and automated project management,
- automated software development and maintenance, integrated development steps and tools, automated generation of software code, and faster software development process,
- software reusability and software portability across environments,
- automated error checking, reduced number of errors, and improved software quality, and
- standardized software documentation, automated generation of software documentation, and documentation reusability.

A complete CASE system has the following characteristics:

- graphics interface for drawing structured diagrams,
- information repository for storing and managing all software system information,
- highly integrated toolset sharing a common user interface,
- tools to assist every phase of the life cycle,
- prototyping tools, and
- automatic code generation from design specifications.

An integrated CASE environment is a system that links, coordinates, and manages the activities, information, and deliverables that are the critical foundation of software engineering in a team environment. The CASE environment accommodates specialized tools through a consistent, friendly interface. It does not necessarily depend on one life cycle methodology or language but can encompass many of each.

21.2 Impact on Ada

Many CASE implementers are beginning to realize the potential in the Ada language. Ada is the first language designed from the beginning to support software engineering techniques throughout the software life cycle, with not only the language but also the run-time and programming environments specified. The existence of the Stoneman definition for the APSE has given an incentive to support the software development in Ada with CASE systems.

Currently, few CASE tools are targeted to Ada systems. There is a trend developing, however, for an increasing number of CASE tools available to support Ada development. Many of the companies that have existing CASE tools are adding an Ada capability to those tools. This is especially true of those companies whose focus is real-time development and those that offer real-time enhancements to their standard packages. There are also new CASE tools being developed with Ada software as the primary focus.

A cross-section of the tools either on the market or under development include:

BCASE The Bendix Computer-Aided Software Engineering system is an environment for large-scale software engineering based on Ada. It utilizes a knowledge-assisted graphical editor for capturing and modifying the software designs expressed in a graphical design language (Ada/GDL).

BYRON ¹⁴ The system is a PDL environment with an integrated document generator and Ada compiler.

IAW The Interactive Ada Workstation develops Ada code from a graphical representation based on Buhr diagrams. (See Section 18.4)

MAESTRO ¹⁵ This system emphasizes the organizational and management aspects of the software life cycle. It provides a three tier system: the target system where compilation and testing occur, the departmental minicomputer for coordination and configuration management, and the development workstation. The Maestro system provides a very detailed real-time project management and time accounting system, giving the program manager a detailed look at the latest performance data on line. This data is collected as part of the standard development procedure and requires little or no extra effort on the part of the developers. Design methods are left up to the development organization's unique standards. Maestro does provide graphics editing, configuration management, standard review cycles, etc., but no rigorous support of individual methods.

SOFTWARE THROUGH PICTURES ¹⁶ This is an open architecture system that

¹⁴Byron is a registered trademark of Intermetrics, Inc.

¹⁵Maestro is a registered trademark of Softlab, Inc.

¹⁶Software Through Pictures is a trademark of Interactive Development Environments.

allows the definition of graphics specification and design techniques and their translation into code templates. It comes with several methods already defined including: Jackson Data Structure charts, Chen Entity Relationship diagrams, Yourdon Data and Control Flow Diagrams, Constantine Structure Charts, Transition Diagrams, State Transition Diagrams, and a Table Editor.

RULETOOL ¹⁷ This is another open architecture tool for graphic methodologies, but the emphasis is more on the syntax and semantic rules of the methods involved. Where Software Through Pictures only allows informal rule checking, Ruletool allows the specification of detailed rules that will be checked by the machine during an interactive session. Many of the standard methods are predefined as above, but there are additions such as program management tools and a data dictionary.

RAPID The Reusable Ada Package for Information Management Development consists of a methodology for the development of reusable software components in Ada as well as a system for the retrieval of reusable components according to user specifications.

STATEMATE This system provides a structural, functional, and behavioral view of a real-time and embedded Ada system. This is a new CASE system developed for Ada systems (especially real-time). It includes an innovative representation technique for real-time dynamics as well as support for many of the standard representation techniques. The first real production version is due to hit the market in the fall of 1987.

Software Technology for Adaptable, Reliable Systems (STARS) will provide tools and environments for building Ada programs on a common base. The STARS program consists of four parts:

- Common Ada Foundations,
- Ada environments,
- Shadow projects, and
- An Ada repository.

The Common Ada Foundation consists of soliciting the development of reusable Ada software modules. The goal is to provide Ada building block components for several foundation areas (such as software design, text processing, operating systems, graphics, etc.).

¹⁷Ruletool is a trademark of The CADWARE Group, Ltd.

The Ada environments consists of building environments based on the Common Ada Foundation. They will be integrated, adaptable collections of methods and tools.

The shadow projects will use the environments to duplicate existing projects in order to demonstrate that the STARS methods can produce programs faster, cheaper, and more reliably than the current methods.

The repository will be an integral part of the STARS program. All deliverables, including documentation as well as code, will be placed in the repository and made available to future software developments.

The European Economic Community is sponsoring a CASE tool which is a software development system based on a denotational semantics metalanguage. (See Sections 17.1.1 and 17.5.1 for a discussion of denotational semantics.) The system has three major components: a software project management system, a project graph, and a library of verified transformations. The nodes in the project graph represent documents, specifications, software modules, etc. Their interconnections are used to show the derivation or construction of the software. The goal of this CASE tool is to enable less highly trained personnel to develop more reliable software by underpinning the environment with the principles of formal methods and verification.

21.3 Environment Standards and CASE

It would be advantageous for future CASE tools to be developed for Stoneman and CAIS environments (see Section 18 for a description of these system interfaces). This would help solve some of the major problems faced by the users of CASE tools today. Some of the biggest problems are:

- There is no interoperability between CASE tools,
- CASE tools cannot be easily transported between processing environments, and
- Many CASE tools cannot support the networking or multi-tier environments required for coordination between developers.

These are precisely the problems addressed by the Stoneman and now by CAIS. As Ada becomes a more important part of the CASE market and as CAIS becomes a more important part of the Ada community, we should see CASE products conforming to CAIS interfaces.

21.4 CASE Activities

Several CASE Symposia have been held to date. In June 1987, a CASE Symposium was held in Washington D.C.. Proceedings of the symposium are available. An information packet was produced by:

Digital Consulting Inc.
6 Windsor Street
Andover, MA 01810
(617) 470-3870

The First International Workshop on CASE was held in May 1987 in Cambridge, MA. Proceedings are available through:

CASE '87
First International Workshop on
Computer-Aided Software Engineering
c/o Index Technology Corporation
One Main Street
Cambridge, MA 02142

Appendix A

References

- [ABB86] Agrawala, A., Bloom, M., Boorstyn, R., Buhr, R.J., Heystek, D., Ada Foundation Technology, Volume 9, Software Requirements for WIS (WWMCCS (World Wide Military Command and Control System) Information System) Network Protocol Prototypes, Institute for Defense Analyses, Alexandria, VA, December 1986.
- [ADETS87] Ada Education and Training Study, Volume 1, AFCEA, Fairfax VA, July 1987.
- [BRF86] Brauer, D.C., Roach, P.P., Frank, M.S., Knackstedt, R.P., Ada and Knowledge-based Systems: A Prototype Combining the Best of Both Worlds, McDonnell Douglas Astronaut. Co., Huntingdon Beach, CA, October 1986.
- [B&N78] Basili, Victor R. and Noonan, Robert E., A Comparison of the Axiomatic and Functional Models of Structured Programming, TR-630, University of Maryland, College Park, Maryland, February 1978.
- [B&R86] Burns, A., Robinson, J., ADDS - A Dialogue Development System for the Ada Programming Language, Bradford University, England, February 1986.
- [Coh86] Cohen, Norman H., "Ada Axiomatic Semantics: Problems and Solutions," SofTech TP-223, Proceedings, Ada-Europe Conference, Edinburgh, Scotland, February 1986.
- [CohN86] Cohen, Norman H., "MAVEN: The Modular Ada Validation Environment," SofTech TP-227, Third IDA Workshop on Ada Verification, Raleigh, North Carolina, May 1986.
- [Dye83] Dyer, M., Software validation in the cleanroom method. Technical Report TR 86.0003, IBM Federal Systems Division, Bethesda, Maryland, August 19, 1983.
- [FCW87] "DOD Plows funds into R&D for Ada Tools", Federal Computer Week, August 24, 1987.

- [FHD86] Foley, J., Hrycyszyn, J.D., Denbrook, P., Gilroy, K., Green, M., Ada Foundation Technology, Volume 8, Software Requirements for WIS (WWMCCS (World Wide Military Command and Control System) Information System) Graphics System Prototypes, Institute for Defense Analyses, Alexandria, VA, December 1986.
- [F&B86] Friedman, F.J., Brykczynski, B.R. Ada/SQL: A Standard, Portable Ada-DBMS Interface, RACOM Computer Professionals, Annandale, VA, February 1986.
- [Han86] Hansom, M. S., Application of Advanced Ada Language Features to Data Structures in a Graphic Programming Environment Master's Thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1986.
- [HVR??] Hood, P.E., Vanderminden, C.A., Ruegsegger, T.B., "Automating IDEF₀ to Ada Translation", SofTech, Inc., Waltham MA, (Unpublished).
- [L&G86] Liskov, B. and Guttag, J., Abstraction and Specification in Program Development, The Massachusetts Institute of Technology, 1986.
- [L&H84] Luckham, David C. and von Henke, Friedrich W., "An Overview of Anna, a Specification Language for Ada," IEEE Computer Society 1984 Conference on Ada Applications and Environments, IEEE Computer Society Press, Silver Spring, Maryland, 1984, 116-127.
- [Mar87] Marmelstein, Lt Robert E., "The Interactive Ada Workstation: A Prototype for Next Generation Software Environments", Proceedings of the Joint Ada Conference, Fifth National Conference on Ada Technology and Washington Ada Symposium, March 16-19, 1987, pp 54-61.
- [Mye79] Myers, Glenford J., The Art of Software Testing, John Wiley & Sons, New York, 1979.
- [M&C??] Munck, R., and Cohen, S., "SAda Project Overview", MITRE Corporation, Bedford MA, (Unpublished).
- [PMP86] Greene, J.S., Jr., Probert, T., Riddle, W., Giese, C., Quindry, T.L., Trimble, J., Software Technology for Adaptable, Reliable

Systems (STARS) Program Management Plan, STARS JPO, August 6, 1986.

[SBB85] Selby, Richard W., Jr., Basili, Victor R., and Baker, F. Terry, CLEANROOM Software Development: An Empirical Evaluation, TR-1415, University of Maryland, College Park, Maryland, February 1985.

[SDME86] System Specification for the Software Development and Maintenance Environment, Prepared by WIS Division, GTE Government Systems, October 29, 1986.

[Sto80] Buxton, J., Department of Defense Requirements for Ada Programming Support Environments "Stoneman", DoD, February 1980.

[TPP86] Greene, J.S., Jr., Probert, T., Riddle, W., Giese, C., Quindry, T.L., Trimble, J., Software Technology for Adaptable, Reliable Systems (STARS) Technical Program Plan, STARS JPO, August 6, 1986.

[Tra87] Tracz, Will, "Ada Reusability Efforts: A Survey of the State of the Practice", Proceedings of the Joint Ada Conference, Fifth National Conference on Ada Technology and Washington Ada Symposium, March 16-19, 1987, pp 35-44.

Appendix B

Bibliography

"Ada's Role in the \$300+ Million Case Market", Ada Data, January, 1987, Volume 5, Number 1, pp 1-16.

Back, R.J.R., Correctness of Explicitly Specified Procedures, Mathematical, Amsterdam, 1980.

Basili, V.R. and Mills, H.D., Understanding and Documenting Programs, Technical Report TR-884, University of Maryland, Computer Science Center, April 1980.

Computer-Aided Software Engineering Symposium, Digital Consulting, Inc. Summer 1987 Edition.

Correll, Claus H., Proving Programs In Several Steps of Refinement, Computer Sciences Department, IBM Thomas J. Watson Research Center, November 8, 1976.

Dahl, Ole-Johan, "Can Program Proving Be Made Practical?", Lectures presented at the EEC-CREST course On Programming Foundations, Toulouse, France, May 1978.

Dunlop, Douglas D., An Investigation of Functional Correctness Issues, Technical Report TR-1135, University of Maryland, Computer Science Center, January 1982.

Elsapas, B, Hare, D.F., Levitt, K.N., and Snyder, D.L., Jovial Program Verifier Rugged Jovial Environment, SRI International, Rome Air Development Center, Griffiss Air Force Base, October 1982.

Foreman, J., and Goodenough, J., Ada Adoption Handbook: A Program Manager's Guide, ESD-TR-87-110, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, May 1987.

Friedman, F., Keller, A., Salasin, J., Wiederhold, G., Berkowitz, M.R., Spooner, D.L., Reference Model for Ada Interfaces to Database

Management Systems, RACOM Computer Professionals, Annandale, VA, February 1986.

Hill, I.D. and Meek, B.L., Programming Language Standardization, Ellis Horwood Limited, 1980.

Jenkins, Joyce R., Automated Generation of Input Output Pairs for the CAIS Validation Test Suite, Doctoral Thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, May 1986.

Kamel, Z., Vines, D.H. Jr., A Virtual Database Interface for Ada Applications, GTE Communication Systems, Phoenix, AZ, February 1986.

Kramer, J.F., Oberndorf, P., Long, J., Robinson, R.M., Roby, C., Chludzinski, J., Clouse, J., The CAIS Reader's Guide, Institute for Defense Analyses, December 1985.

Manley, Dr. John H., "Computer Aided Software Engineering (CASE) Foundation for Software Factories", IEEE, 1984, pp 84-91.

Maurer, W.D., Introduction to Programming Science, Part II: Proofs of Assertions About Programs, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, August 21, 1973.

Mills, H.D., Basili, V.R., Gannon, J.D., Hamlet, R.G., Principles of Computer Programming, A Mathematical Approach, Allyn and Bacon Inc., 1987.

Mills, H.D. and Linger, R.C., "Data Structured Programming: Program Design without Arrays and Pointers", IEEE Transactions on Software Engineering, Volume SE-12, Number 2, February 1986, pp 192-197.

Schmidt, David A., Denotational Semantics a Methodology for Language Development, Allyn and Bacon, Inc., 1986.

Schwartz, J.H., "The Bendix Computer-Aided Software Engineering System: A New Approach to an Ada Design Language", IEEE, 1986.

Schwartz, R.L., Melliar-Smith, P.M., The Suitability of Ada for Artificial Intelligence Applications, SRI International, Menlo Park,

CA, May 1980.

Sparks, M.R., Gallop, J.R., Language Bindings For Computer Graphics Standards
SDC/Burroughs, Huntsville, AL, August 1986.

Stamps, David, "CASE: Cranking Out Productivity", Datamation Magazine,
July 1, 1987, pp 55-58.

Stoy, Joseph, The Scott-Strachey Approach to the Mathematical Semantics
of Programming Languages, Massachusetts Institute of Technology
December 1974.

Waychoff, Richard, "The Stars Program Today", CASE '87 First International
Workshop on Computer-Aided Software Engineering, Advance Papers,
Volume 1, May 27-29, 1987, pp 492-494.

Appendix C

Points of Contact for Ada Information

Ada Board - Ada Technology and
Standards Panel

Bill Carlson
Intermetrics, Inc.
4733 Bethesda Ave.
Suite 415
Bethesda, MD 20814

Ada/SQL Proposed Binding

Lt Col Terry Courtwright
WIS JPMO
Washington, D.C. 20330
(703) 285-5067
court@MITRE.ARPA

Ada/SQL Working Group (ASWG)

Bill Brykczynski
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria, VA 22311
brykczyn@AJPO.SEL.CMU.EDU

Ada Test and Verification System
(ATVS)

Deborah A. Cerino
RADC/COEE
Griffiss Air Force Base, NY 13441
(315) 330-2054
AUTOVON 587-2054
cerino@RADC-SOFTVAX.ARPA

Ada Validation Office (AVO)

Audrey Hook
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria, VA 22311
(703) 824-5501

ADETS Report

Brig. General, Kirby Lamar, (Ret.)
Director, Corporate Affairs
AFCEA
4400 Fair Lakes Court
Fairfax, VA 22033-3899
(703) 631-6235

Automated Measurement System
(AMS)

Roger J. Dziegiel, Jr.
RADC/COEE
Griffiss Air Force Base, NY 13441
(315) 330-2054
AUTOVON 587-2054
dziegiel@RADC-SOFTVAX.ARPA

CAIS Revision A

Patricia Oberndorf
Code 423
Naval Ocean Systems Center
421 Catalina Boulevard
San Diego, CA 92152-5000
(619) 225-6682/7401
AUTOVON 933-6682/7401
oberndor@AJPO.SEI.CMU.EDU

CAIS Implementation Validation
Capabilities

Ray Szymanski
Air Force Wright Aeronautical Laboratories
AFWAL/AAAF-2
Dayton, OH 45433
(513) 255-2446
AUTOVON 785-2446
rszymanski@ADA20.ISI.EDU

CASE Symposium
Washington D.C., June 1987

Digital Consulting Inc.
6 Windsor Street
Andover, MA 01810
(617) 470-3870

CASE
The First International Workshop
Cambridge, MA, May 1987

CASE '87
First International Workshop on
Computer-Aided Software Engineering
c/o Index Technology Corporation
One Main Street
Cambridge, MA 02142

Computational Logic

Michael Smith
Computational Logic
1717 West 6th, Suite 290
Austin, TX 78703
(512) 322-9951

Dansk Datamatik Center

Hansen, Kurt W.
Dansk Datamatik Center
Lundtoftevej 1C
DK-2800 Lyngby
Denmark
+45 2 87 26 22
khansen@ADA20.ISI.EDU

Electronic Systems Division (ESD)
Security Analysis of Ada Programs

Lt John N. Molloy
ESD/SYC-2
14 Oak Heart
MITRE Building L
Bedford, MA 01730
(617) 271-5053

Institute for Defense Analyses (IDA)
Ada Verification

Terry Mayfield
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria, VA 22311
(703) 824-5524

Interactive Ada Workstation (IAW)

Lt Robert Marmelstein
AFWAL/AAAF
Air Force Wright Aeronautical Labs
Wright-Patterson Air Force Base, OH 45433-6543
(513) 255-6548/3947
AUTOVON 785-6548/3947
rmarmelstein@ADA20.ISI.EDU

PolyAnna

Don Elefante
RADC/COTC
Griffiss Air Force Base, NY 13441
(315) 330-3241
AUTOVON 587-3241
elefante@RADC-MULTICS.ARPA

Revision of Ada Investigation

Dr. John Kramer
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria, VA 22311
(703) 845-2263
AUTOVON 289-1948 ext 2263

SDME

Capt James B. Hogan
Headquarters
Electronic Systems Division
ESD/SYW-2P1
Hanscom AFB, MA 01731-5000
(617) 377-4754
AUTOVON 478-4754
hogan@MITRE.ARPA

SIGAda - Formal Methods Committee

Richard A. Platek
Odyssey Research Associates, Inc.
1283 Trumansburg Road
Ithaca, NY 14850-1313
(607) 277-2020
rplatek@Ada20.ISI.EDU

Software Life Cycle Support
Environment (SLCSE)

Frank S. LaMonica
RADC/COEE
Griffiss Air Force Base, N.Y. 13441
(315) 330-2054
AUTOVON 587-2054
lamonica@RADC-SOFTVAX.ARPA

STARS

Col Joseph S. Greene, Jr.
Director, STARS
STARS JPO
Office of Secretary of Defense
OUSDRE (R&AT/CET),
The Pentagon, Rm. 3E114
Washington, DC 20301
(202) 694-0210